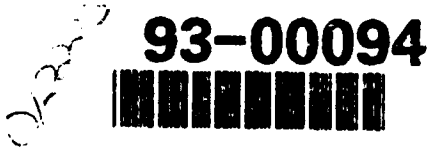
AFIT/GCE/ENG/92-D-07

AN INVESTIGATION OF DISCOVERY-BASED LEARNING IN THE ROUTE PLANNING DOMAIN

THESIS

Freeman A. Kilpatrick Jr.
Captain, USAF

AFIT/GCE/ENG/92-D-07



Approved for public release; distribution unlimited

AFIT/GCE/ENG/92-D-07

AN INVESTIGATION OF DISCOVERY-BASED LEARNING IN THE ROUTE PLANNING DOMAIN

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Freeman A. Kilpatrick Jr., B.S.E.E., M.S.S.M.

Captain, USAF

December, 1992

Acknowledgements

I would like to express my appreciation to my thesis advisor, Maj Gregg Gunsch, for giving me the freedom to research what I thought was important, and for his innocent questions that usually resulted in many hours of extra work. I would also like to thank my committe, Dr. Thomas Hartrum and Dr. Henry Poctozny for their contributions to this effort.

I also must thank my parents, Freeman & Sara Kilpatrick for instilling in me a love for learning that has carried me through endless hours of school, and for continually giving me unbounded support, confidence and love.

Finally, this thesis would not have been possible without the support of my wife Donna. Her love, encouragement, and confidence in me were every bit as essential to the effort as a SPARCstation.

Freeman A. Kilpatrick Jr.

Table of Contents

List of Figures

List of Tables

Abstract

This thesis presents MAVERICK, a Discovery-Based Learning (DBL) system designed to learn maneuvers in the route-planning domain. DBL was originally designed to learn in domains for which little domain knowledge exists. This thesis proposes using it in domains for which knowledge exists, but the acquisition of this knowledge is difficult or time-consuming because of the knowledge acquisition bottleneck.

The operation of the DBL process in MAVERICK was investigated to determine the potential utility of such a system to a real-world Air Force problem in the domain of aircraft route planning. MAVERICK worked well in its limited domain, and demonstrated several positive aspects of the DBL process, specifically robustness, flexibility, and graceful degradation. Some negative aspects of the process were also encountered during this research; MAVERICK demonstrated a pronounced tendency towards unpredictability, both in its operation and its development. This likely precludes DBL from application to critical systems, however the positive aspects suggest DBL can have potential utility to other systems.

AN INVESTIGATION OF DISCOVERY-BASED LEARNING IN THE ROUTE PLANNING DOMAIN

1. Introduction

1.1 Background

In a modern fighter cockpit, a pilot has to deal with an ever-increasing burden on his mental faculties as aircraft systems grow in complexity and number. In a two-seater aircraft, a co-pilot can relieve some of the pressure from the pilot, but the added weight and burden to the aircraft frequently makes this an unacceptable solution. In 1986, the Pilot's Associate (PA), a joint DARPA/USAF program targeted to develop technology in the area of real-time, cooperating knowledge-based systems, was begun to address this problem. The overall objective of the program was to develop an "intelligent" automated assistant to a pilot of a modern advanced fighter aircraft. The assistant obtains and compiles information, recommends courses of action, and in some cases actively performs mission tasks. One of the key elements of PA is the Tactics Planner (TP). This module reacts to the dynamic, tactical environment by responding to immediate threats and determining their impact on the mission (1).

One of the problems with implementing a large, complex, knowledge-based system such as the PA is the sheer size of the domain-specific knowledge base. An enormous amount of human expertise must be coded into a form usable by the PA system. Also, it is difficult to translate a pilot's knowledge into the specialized, specific forms needed for the PA. The typical method of translation involves having the pilot express his knowledge to a ***knowledge engineer***, who then translates it into a form understandable by the computer. This process can be extremely time-consuming, providing a significant negative impact to the development time of the system, as well as potential translation difficulties or misunderstandings. This problem is a frequent hurdle

in Artificial Intelligence (AI) systems, and is referred to as the ***knowledge acquisition bottleneck***. Another difficulty is the maintenance of the knowledge base; as new aircraft technology or tactics develop, major portions of the knowledge base must be modified and updated. Finally, the size of the knowledge base makes accuracy a prime concern. It may be difficult to guarantee 100% accuracy of the knowledge base, but errors in it can cause potentially lethal malfunctions of the PA system.

To help solve this problem, the Wright Research and Development Center (WRDC) began the Learning Systems for Pilot Aiding (LSPA) program. The objective of this program was to demonstrate the feasibility of using machine learning to automate the process of incorporating new information into the knowledge bases of the PA. Machine learning (ML) is a sub-field of AI that deals with automating the acquisition of new knowledge by an AI system, with the goal of improving the performance of the system. Although machine learning cannot completely eliminate the knowledge acquisition bottleneck, it has the potential to greatly reduce it by cutting out the main source of the bottleneck, the knowledge engineer. If the computer can obtain information on its own, then the information is already in a usable form, and needs no translation. If the computer can learn from its (or others') experiences, then it can improve over time and become self-correcting, compensating for computer or human induced errors.

1.2 Discovery-Based Learning

The machine learning technique chosen for LSPA (Explanation-Based Learning) is very knowledge-intensive, and relies strongly upon this knowledge to be able to learn. To investigate alternative approaches which require less knowledge, Discovery-Based Learning (DBL) was chosen for this thesis. DBL is a relatively novel approach to machine learning, and is characterized by the fact that it is ***unsupervised***: a DBL system attempts to discover concepts without any human guidance. The basic concept behind DBL is to mimic the scientific method: study a domain, gather

data, formulate hypotheses, test with experiments, and repeat. A DBL system will have access to some method of independently gathering data about the given domain and performing its own "experiments".

1.3 Problem

The problem to be investigated is how to improve the performance of knowledge-based systems by partially automating the process of knowledge acquisition. This thesis proposes to automate more of the process than conventional means, by using DBL to allow a system to learn pilot tactics on its own.

1.4 Approach

This study uses RIZSIM, a general purpose combat environment simulator developed for parallel processing research at the Air Force Institute of Technology (AFIT). RIZSIM will act as the "world" upon which the DBL system will perform its experiments. This simulator is a reasonable fidelity model, with a significant amount of flexibility built into the underlying code. The core of this thesis is the development of a system which can use the simulator to fly experimental maneuvers, discover the best maneuver for a scenario, and learn about maneuvers as the program progresses.

1.5 Scope

Because of hardware and software availability, this study will be limited to scenarios involving a single aircraft and single or multiple SAM sites. No multiple aircraft scenarios will be investigated. It is also understood that this process is limited by the fidelity of the simulator.

1.6 Objectives

The overall objective of this thesis is to investigate the feasibility of applying unsupervised learning to the domain of pilot tactics. As part of this process, the following objectives have been accomplished:

- Determine if the domain knowledge "built-in" to a simulator is sufficient to guide a discovery process.
- Determine if the maneuvers generated are of sufficient generality to apply to a variety of scenarios.
- Determine how the limiting effect of the simulator fidelity affects the generated maneuvers.
- Determine if a discovery process can generate maneuvers that are applicable to real life
- Characterize the performance of the DBL process..

II. Literature Review

2.1 Background

2.1.1 Why Machine Learning? Because of the complexity of an advanced fighter aircraft and the complexity of the pilot's activities, the knowledge required for an associate system is formidable. Miller (14) states that in a domain as complex as tactical aviation, it would be impossible to anticipate all the important decisions facing a pilot in future situations. Thus, it would be exceedingly difficult to build a knowledge base that could cover all the issues of interest to pilots. He also highlights the significant issue that static knowledge bases are implicitly incompatible with the dynamic, evolving changes in enemy capabilities and tactics. Finally, he stresses the labor intensive nature of knowledge acquisition. The PA will require expertise from a wide variety of experts, including pilots, systems engineers, tacticians, human factors engineers, and display engineers. The reason behind the requirement for such a wide spectrum of expertise is the synergy of the pilot and the PA system. For example, displays may change to accommodate different tactical plans, requiring human factors expertise. Aircraft systems may adjust to improve the performance of the aircraft for a particular plan, requiring systems engineer expertise. Because of the number of experts required, modifications of tactical plans in the knowledge base will be a costly, time-consuming task; modifications during force deployment may be impossible (14).

Miller states that ML has a high payoff for application to the PA. Some of the more significant benefits include: knowledge acquisition automation, increased speed and reliability of modifications, less down time, and a reduced need for domain experts. Lizza, *et al*, (1) even state the PA may be impossible without ML.

One division of machine learning methods is the division between deductive and inductive systems. Deductive systems operate by deducing relationships between knowledge that is already in the system. They operate on the transitive closure of knowledge, the sum of knowledge in the system that is represented explicitly or can be deduced (18). Inductive learning systems typically

deal with classification. Given a set of examples, they attempt to induce a concept which will distinguish between elements of the examples. This is done with some stimuli from the outside world, typically in the form of a teacher who classifies the examples as positive or negative. The system learns how to differentiate the positive from the negative. Some inductive systems, however, can learn without using a teacher to classify the examples.

This literature review will investigate modern research in the field of machine learning with an emphasis on the applicability of these methods to a PA-type system. Specifically, the emphasis will be on the application of learning systems to the domain of pilot tactics. The review will concentrate on two main learning schemes: deductive and inductive.

2.1.2 Machine Learning Definitions One of the problems with classifying machine learning techniques is the lack of a formal definition for machine learning. Many authors' definitions directly conflict with other authors' definitions; the perception of what constitutes learning influences the actual implementation of various learning techniques. For example, Dietterich (4) defines learning as an simply an *increase* in knowledge. In direct contrast, Fagin and Halpern (5) define learning as the translation of implicit beliefs into explicit beliefs. Under Dietterich's definition, Fagin and Halpern's system does not constitute learning because it does not involve an *increase* in knowledge; the knowledge was already present in the system implicitly. Rick and Knight (18) present a more pragmatic definition. They simply define learning as the ability of a system to improve its performance of a given task after its first performance of the given task. Because of the unifying nature of this definition, it will be implied whenever learning is discussed throughout this thesis.

2.2 Deductive Learning

Explanation-Based Learning (EBL) is the primary form of deductive learning. The basic EBL process is describe by DeJong (3). The two main players in an EBL system are the learning system and an agent that acts as the teacher for the learning system. The EBL system observes

the behavior of the agent, then reacts when the agent's actions deviate from the learning system's understanding of the world. Sometimes the agent's actions achieve a goal that is not expected by the the learning system, then "the system must construct an explanation or proof of how the goal was achieved from the agent's actions using a domain theory of how actions affect goals in that domain."(3) The learned explanation is stored for future use, improving the performance of the system over time.

Mitchell (15) provides a description of the inputs to an EBL system:[1]

Goal Concept A concept definition describing the concept to be learned.

Training Example An example of the goal concept.

Domain Theory A set of rules and facts to be used in explaining how the training example is an example of the goal concept.

Operationality Criterion A predicate over concept definitions, specifying the form in which the learned concept must be expressed.

An example of this is presented by Levi (12). In this example, the EBL system learns the doppler notch maneuver (12). The doppler notch is a real-world pilot tactic which simply requires the pilot to fly at a constant radius around a SAM site, thus denying the SAM site doppler range data. The first input to the EBL system is the learning instance, which in this case is a simulator trace of the pilot flying a doppler notch. The second input is the pilot's goal, in this case "safe-from-sam-site." This is the target concept that the system should learn. The third input to the system is the domain theory that was previously a part of the system. Using this domain theory, the system cannot explicitly prove the accomplishment of the goal using the learning instance, so it deduces a proof by searching through the domain theory. This proof is then passed to an

[1]Explanation-Based Generalization (EBG) is a form of EBL that learns a generalized concept. It is essentially used synonymously with EBL in modern literature.

"Explanation Generalizer", which removes the scenario specific aspects of the new concept, and produces a generalized plan that can be used in future scenarios.

Deductive learning is built to take advantage of domain theory (18). This ability to leverage the knowledge present in the system can provide several advantages. Primary among these is the ability for the system to justify its deductive explanations. Since the domain knowledge is represented explicitly, the system can justify its new knowledge by presenting to the user a logically consistent chain of knowledge that is already present in the system (4). Another advantage to deductive systems is their guarantee of correctness. If a deductive system has perfect domain theory, then it is guaranteed to provide correct explanations (15). Miller (14) states that EBL systems make the best use of single learning experiences. This is a significant advantage in the pilot tactics domain, where multiple examples of a target concept may not be available.

Deductive methods also have several disadvantages. One disadvantage is the problem of determining an appropriate level of generalization for the explanations. Ideally, an explanation is general enough to apply to situations that are similar to the learning instance, but not identical. If the generalization is too specific, the the system will soon become cluttered with a large set of ungeneralized problem-solution pairs (4). If the generalization is too general, then the system may apply the learned explanation to an inappropriate situation. Either of these can quickly degrade the performance of a knowledge based system. Second, deductive systems are fairly brittle; they are not able to handle imperfect domain theories well (18). Related to this problem is the difficulty of applying deductive methods to domains that are not easily formalizable (18, 7). A selective over-generalization method can be used to apply deductive methods to address the problem of imperfect domain theories. Abduction can also be used to generate potential hypotheses to eventually arrive at a potentially correct solution (4). However, both of these methods can compromise the guaranteed correctness of the deductive process. Finally, deduction, by its very nature, does not allow for the application of knowledge in novel ways (9).

2.3 Inductive Learning

Paul Scott, in Levi (13), defines an inductive learning system as "one which uses experience to generate a conjecture and then proceeds to use further experiences to confirm or refute that conjecture."

One of the most well known examples of an inductive learning system is Quinlan's ID3 (17). ID3 learned descriptions of classes; given positive and negative examples of a concept, it discovered methods of distinguishing between the positive and negative examples, using decision trees.[2] Once an induction program has learned a concept from a training set of examples, it can be applied to a new set of examples for classification.

An intereresting feature of inductive learning is the ability of the system to generate knowledge that was not present in the system implicitly or explicitly before the learning began (4, 13). Inductive learning is based on the idea that a large part of learning for any system (human or machine) involves classification (18). One of the primary advantages of inductive learning is that it requires very little domain theory. Ideally, it would only require a sufficient set of training examples, and an algorithm for determining a classification method. The system can potentially learn the concept represented by the examples even if no domain theory exists anywhere. Although inductive systems do not have domain theory like deductive systems, they do have to deal with imperfect learning information. In inductive systems, false or erroneous examples are usually referred to as *noise*. Given enough examples, the inductive system has the ability to isolate noise from the good data, giving it much greater robustness than a strictly deductive system (18).

While inductive systems provide a significant amount of flexibility, they also suffer from some significant shortfalls. It is often difficult to define and quantify which features of a given example are significant. This makes the development of efficient inductive learning algorithms difficult. In fact, Dietterich (4) states that "there are no efficient, general-purpose inductive learning methods."

[2]Examples are usually represented as sets of attribute-value pairs.

Another problem with inductive systems is the proper selection of training examples. To teach a given concept, the order and appropriate mix of examples in the early training stage can be critical (7). Finally, there may be some domains where it is impossible to obtain appropriate examples (11).

2.4 Discovery-Based Learning

One form of inductive learning is Discovery-Based Learning (DBL), which is often referred to as *inductive inference* (7). This type of learning is quite different from a system which learns from examples, however. DBL systems do not operate with the aid of a teacher/classifier or a set of training examples. Instead, DBL systems learn by experimentation. This type of learning is ideally suited to complex, dynamic applications requiring reactive feedback (2).

Lenat (7) has been the primary developer of DBL systems. He is the author of AM[3], a program used to discover new concepts in the domain of mathematics. Starting from basic set theory, it discovered many significant mathematical concepts such as multiplication as repeated addition, prime numbers, and Goldbach's conjecture.[4] It discovered many other concepts that were unknown to the author, but not unknown to mathematicians.

2.4.1 Heuristics The primary element of a DBL system is its set of heuristics.[5] The standard definition of a heuristic is a "rule-of-thumb", a rule which is useful but not guaranteed to be correct. Rich and Knight (18) define heuristics in terms of AI and search, by "a technique that improves the efficiency of a search process, possibly by sacrificing claims of completeness (of the search)." In a search, heuristics may help guide the system to quick solutions, but may also miss other potentially better solutions. Lenat (7) proposes an unusual definition of heuristics as "compiled hindsight." This definition is interesting because it highlights some of the power of heuristics.

[3] AM originally stood for Automated Mathematician, but this meaning has been dropped.
[4] This is the unproven theorem that all even numbers can be expressed as a sum of exactly two primes.
[5] The word *heuristic* comes from the Greek word *heuriskein*, meaning "to discover" (18).

Because there is much continuity in the world, heuristics which work in a particular problem can be applied to many similar problems. Heuristics are able to encapsulate a large chunk of knowledge about how the world works into a compact unit. Although this is shallow knowledge, [6] it can still be quite useful in cutting down a search space.

An example of a heuristic used in AM was the "look for extremes" heuristic. While AM was exploring the concept "numbers with n divisors", it looked at zero and one (finding no examples), and then "numbers with only two divisors", or prime numbers. While it may have looked at numbers with three or four divisors, it did not continue to and explore numbers with twenty, thirty, forty, etc. divisors. Interestingly, this heuristic can be applied to numerous domains quite far removed from number theory.

AM required 243 heuristics to accomplish what it did. This has led to the obvious criticism that perhaps having such a large body of heuristics "led" AM directly to the discoveries desired by the author (11). This is certainly a valid criticism, but it does not explain how AM was able to "discover" concepts that were unknown by its author. In a study of AM, Ritchie and Hanna (19) bring up the significant point that extreme care must be taken in how the performance of an AI program is interpreted.

2.4.2 DBL Operation The general operation of DBL is an inductive process: "constrain attention to a manageable domain, gather data, perceive regularity in it, formulate hypotheses, conduct experiments to test them, then use the results as new data with which to begin the cycle again." (9)

2.4.2.1 The Agenda Central to any DBL system is the *agenda*. The agenda is the structure which holds the list of tasks for the DBL system to perform, listed in descending order of *interestingness*. Interestingness is a representation of how likely the task is to produce interesting

[6] *Shallow knowledge* refers to the fact that there is little or no understanding of the knowledge behind the heuristic. The system has the knowledge that the heuristic works, but not *why*. Expert systems operate almost exclusively with shallow knowledge.

results. It is obviously a subjective[7] measurement, but it can provide significant power to prevent the system from wasting time on inefficient exploration. The basic object of a DBL system is a concept, or conjecture. The heuristics deal with manipulations of concepts such as generalization, specialization, generation of new concepts, and pruning of concepts.

During operation, the DBL system performs the first task on the agenda (the most interesting), which may or may not generate more tasks or concepts. After this task is performed, the heuristics may suggest new concepts to be explored or tasks to be performed. These new tasks are also placed in the agenda in order of interestingness. Then the system repeats the process, performing the new task at the top of the agenda, and so on. The process could typically continue indefinitely, but will usually be halted after a time limit, or when the tasks on the agenda drop below a certain interestingness threshold.

2.4.2.2 The DBL World DBL also requires a world in which to perform experiments. While this could feasibly be the real world, this would require humans to carry out the DBL system's proposed experiments, then report back the results (9). The easier method is to supply the DBL system with a simulated world. The simulated world can be limited to the domain in which the learning system operates. Because the simulated world (simulator) is computer-based, the interaction between the learning system and its world happens at computer speeds. The learning system can perform experiments, and obtain results, limited only by the execution time of the simulator and the learning system.

A DBL system is limited, however, by the fidelity of its simulated world. As an example, Lenat (9) hypothesizes a DBL system designed to explore concepts in the domain of physics. If a Newtonian simulator is used with such a system, it will only be able to rediscover Newtonian physics, and will never be able to discover modern physics or relativity because its world genuinely *is*

[7] Obviously, nothing in a computer can be truly subjective. The subjectivity comes from the human biases used in programming the interestingness evaluations.

non-relativistic. In the real world, whenever humans reach the "boundary", there is usually another level of complexity to discover behind it. The need for a simulator restricts DBL to systems which are internally formalizable, thus eliminating a large section of potential learning domains. This also restricts DBL to domains which can be simulated in a static, "batch" type mode. During the course of exploration, DBL needs to perform experiments on an unchanging world so there is a logical connection between two experiments that were performed at different times. If the world is continually changing, as in a real-time, interactive simulation, then the scope of the DBL experiments is severely limited.

2.4.3 DBL Strengths & Weaknesses One of DBL's primary strengths is that it can be applied to unexplored domains that do not have any developed domain theory (8). Because this type of learning does not rely on domain theory, it can greatly reduce the knowledge acquisition bottleneck that influences both conventional AI systems and deductive learning systems. In fields without human experts, this bottleneck can be so severe as to eliminate the feasibility of any system that cannot learn in an unsupervised manner (9). Finally, because of the inductive nature of the process, a DBL system has the potential for discovering knowledge that has never been discovered by humans, simply because the system may explore certain concepts that humans have never thought to explore because of their own set of biases (9).

Discovery learning has its own particular set of disadvantages. The advantage of potential new discovery also translates to a potential disadvantage. The system must have the proper heuristics to guide its exploration of the world, or it will waste time exploring uninteresting concepts (9). This is a complex trade-off; if the heuristics are too specific, then the system searches a much smaller space, but may not find potentially better solutions outside the space. If the heuristics are not specific enough, then the system may take too long to find solutions. Also, for a discovery system to even begin learning, it must have some method of performing experiments on its environment, or a simulation of its environment. This limits the applications of discovery learning to domains which

are internally formalizable (9). These systems also suffer from problems of interpretability and ratification[8] (13). In many domains, it may be difficult to tell if the learned concept is consistent, correct, useful, or interpretable. The lack of an explicit explanation for the learned concept may be a significant problem in gaining user acceptance. Finally, it is often difficult to assess the results of a discovery learning system. Because of their strong reliance on human-type heuristics to guide the discovery process, it is sometimes difficult to determine what was learned by the system and what was already present in the heuristics (18, 19).

2.5 Conclusion

This chapter highlighted the diverse nature of machine learning systems. Each aspect of machine learning has its own set of advantages and disadvantages; no one method stands out as a generally superior method. However, some methods are better suited to particular applications than others. For the PA-type application, a deductive learning system was chosen primarily because of its predictability and guarantee of correctness. This is certainly a wise move for a system upon which human lives may be depending. However, if guaranteed correctness is not absolutely required, DBL has the potential for significant savings by eliminating the knowledge acquisition bottleneck, and providing a more automated form of learning. This makes it an interesting avenue of research for further exploration of how this process works in practice.

[8] It may be difficult to interpret the concepts learned by the system, and it may be difficult to verify that they are valid.

III. Methodology

3.1 Overview

This chapter details the design considerations that were a critical part of the development of MAVERICK, the prototype DBL system developed for this thesis. MAVERICK is intended to be a proof-of-concept implementation of a DBL system applied to an Air Force problem, for the purpose of learning more about how DBL performs.

There were three primary questions I addressed during the initial development of MAVERICK; the answer to each strongly influenced the remaining questions:

- Which learning domain has applicability to Air Force problems, and is reasonably scoped for a thesis effort?
- Which machine learning method is the most interesting to apply to this domain?
- Which implementation language will work best for this machine learning method?

3.2 Learning Domain

Humans are very generalized learning systems. However, the current state of technology for machine learning requires that the learning system be confined to a small subset of the world, its *domain*. For MAVERICK, I chose the "pilot tactics" domain, to parallel the development of machine learning under the Learning Systems for Pilot Aiding program (LSPA). However, pilot tactics is a very broad domain, so for the purposes of this thesis, the domain was further restricted. MAVERICK is a system which learns tactics in the domain of ***route planning***.

In this context, route planning involves choosing the best path around a set of threat objects (SAM's, aircraft, etc.), minimizing detection by threats while providing the most fuel-efficient[1] route

[1] In reality, the route is the most *time-efficient*. Because the simulator, RIZSIM does not do fuel use calculations, the route that takes the least time uses the least fuel. In a real aircraft, different maneuvers within a route will use different amounts of fuel

possible. This domain was chosen because it has real-world applicability to many AI type problems, as well as the Pilot's Associate (PA). It is also within a reasonable scope for a "proof of concept" domain, and is well understood by humans. A particular difficulty arises when attempting to learn in a domain that is not understood by humans. This difficulty is in knowing whether the learned concepts generated by the machine are correct, false, or the sign of a program deficiency. Because of the nature of machine learning, the machine will likely come up with unexpected conclusions; if the domain is not understood, it may be impossible to judge the worth of these conclusions when they deviate from expected behavior. In route planning, unexpected routes can easily be judged quantitatively to determine their utility.

3.3 Machine Learning Method

The type of machine learning I chose for this effort was Discovery-Based Learning (DBL). There were several factors which influenced this decision; the primary considerations were DBL's small requirement for domain knowledge, its flexibility, and its relative novelty.

3.3.1 DBL and Domain Knowledge A DBL system requires virtually no domain knowledge, it only requires a method of interacting with its world, and heuristics to guide its search. I felt that the knowledge-intensive nature of EBL (used under the LSPA program) represented a significant overhead which would effectively nullify many of the gains to be made from learning. An EBL system is only as good as its domain knowledge; it must use this knowledge to deduce the "why" of the training instance. This knowledge base is subject to all of the problems such as knowledge maintenance, knowledge representation, and the knowledge acquisition bottleneck that motivated research into machine learning in the first place. In short, I felt that EBL was not a big enough step into the area of learning.

Humans frequently learn in limited knowledge domains; scientific exploration is formulated around the discovery of knowledge in a new domain. Of course, for humans, learning knowledge

from a teacher is more efficient than discovering it (i.e., re-inventing the wheel). However, to understand the teacher requires an incredibly large background of supporting knowledge to fill in the gaps of what the teacher does not explicitly teach. For a computer, it may be impossible (and would certainly be inefficient) to represent the background knowledge that humans develop over their lifetime, although research into this area is currently in progress (10). The difficulty lies in knowing *which* elements of background knowledge are relevant. For example, in the pilot tactics domain, knowledge about flower arranging is probably not significant, but knowledge about how a cat stalks a mouse could be quite significant. DBL bypasses the need for knowledge, and simply learns from its environment. This is analogous to a child learning not to touch a hot stove: he doesn't understand *why* the stove is hot, but the knowledge to not touch it serves a very useful function nevertheless.

Pilot tactics is certainly not a new domain; the concepts involved in aerial tactics are relatively *well understood*. However, just because this knowledge is known by humans doesn't mean it can be easily translated to the computer, as previously discussed. I felt that DBL could have a significant payoff, *even in well-understood domains*, because it did not need knowledge, and would not have to be burdened by all the problems associated with knowledge acquisition. A DBL system could learn tactics, without having to understand *why* they worked.

3.3.2 DBL Flexibility Another significant advantage of DBL is its flexibility. Since DBL performs experiments on its own simulated world, that world can be changed simply by changing the simulator. Lenat (7) states that a DBL system is trapped in the world that the simulator defines. While this is true, it does not preclude the discovery of useful concepts. A DBL system is limited by the fidelity of its simulator, but an EBL system is limited by the fidelity of its domain knowledge. The difference is that the simulator developer does not have to understand pilot tactics to design a high-fidelity simulation, but the knowledge engineer for an EBL system does need to thoroughly understand pilot tactics to design a high-fidelity knowledge base.

3.3.3 DBL Research Novelty While researching machine learning, I noticed a significant lack of research involving DBL. DBL was pioneered by Doug Lenat (7), but little follow-up research has surfaced to further explore this aspect of machine learning. The more conventional forms of machine learning have been extensively researched, improved, modified, and critiqued. Given the potential benefits discussed above, this made the investigation of DBL research even more interesting and unique.

3.4 Language

Because MAVERICK is a prototype system, and not intended to be a fielded system, certain flexibility was allowed in the selection of an implementation language. The two primary concerns were speed of development, and portability. Speed of development was needed because the system would be going through many radical, complete design changes as rapid-prototyping progressed because of lessons learned during the development. Portability was a requirement because of the potential for transfer of the system to other researchers. These two factors naturally pointed to the LISP language. The "style" of LISP and the fact that it is interpreted make development extremely rapid when compared to other languages. There is no edit-compile-execute loop as is required with a conventional compiled language. In LISP, a function can be quickly changed, then loaded into the interpreter without any compilation. Portability was addressed through the use of Common LISP, which is a standard developed by the X3J13 subcommittee of ANSI committee X3 (22). Finally, LISP supports the Common LISP Object System (CLOS), to allow for well-structured program development.

There are a few disadvantages to using LISP, however. First is the potential interface difficulty between RIZSIM and MAVERICK, because RIZSIM is written in C, and MAVERICK is written in LISP. LISP has extensive foreign-language interface capabilities, so this did not present a problem. There is also a potential problem with using LISP instead of Ada, because Ada is mandated by

the DOD for all developed systems. Since MAVERICK is strictly a prototype system, this was not judged to be significant in comparison to LISP's advantages.

3.5 Architecture Development

In a DBL system, there are two main architecture considerations: the learning system itself, and the simulated world upon which it can perform experiments. A DBL system could potentially perform experiments upon the real world, but it would probably have to utilize a human to carry-out its experiments (9). This has obvious disadvantages, so only simulations were considered.

3.5.1 Simulation Architecture The main consideration for a simulator for a DBL system is the interface between the DBL system and the simulation. This interface consists of two key elements: the ability to control the simulation, and the ability to get appropriate result data (feedback) from the simulation. Most simulations are written for a specific purpose, and attempting to adapt them to some other purpose is typically quite difficult; typical simulations are not written for generality or ease of modification. The RIZSIM simulation, however, was written at AFIT for just this purpose.

RIZSIM is a general-purpose battlefield simulation written as part of Capt. Robert Rizza's Master's Degree thesis at AFIT in 1990 (20). The original purpose of this model was to provide a "representative" battlefield simulation that could be ported to parallel computing research at AFIT (21). As part of this objective, the simulation was written to be easy to understand and modify, making it ideally suited to a DBL application. The simulation was written in object-oriented style ANSI C, making modifications and interfacing relatively simple.

3.5.2 Learning Architecture Central to the architecture of a DBL system are ***concepts*** and ***heuristics.*** A DBL system manipulates and discovers concepts, using guidance from its heuristics. The two main design decisions for MAVERICK addressed how these would be formulated.

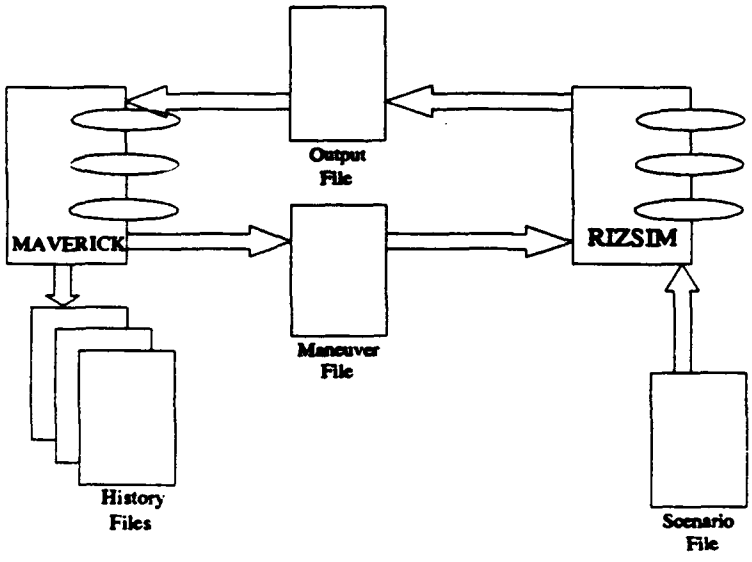


Figure 3.1. MAVERICK Architecture

3.5.2.1 MAVERICK Concepts One of the most basic concepts discussed in terms of pilot tactics is often the "maneuver". This was a natural concept for MAVERICK; I planned for it to explore "maneuver space," or the space of all possible maneuvers for a given scenario. At a high level, a maneuver in MAVERICK can be thought of as a series of turns in response to some threat. A variable length set of turns defines a maneuver. A set of maneuvers defines a "solution" to a given scenario.

It is important to note that the concepts in MAVERICK are maneuvers, and not routes.[2] A route is generated, but this is a consequence of the maneuvers implemented in a given scenario. A maneuver is implemented as a response to a threat object. This is an important distinction; a conventional route planner generates routes that are specific to a given scenario. MAVERICK learns maneuvers that are applicable to a wide variety of scenarios.

3.5.2.2 MAVERICK Heuristics The heuristics in MAVERICK are general-purpose guidelines to influence the direction of the discovery. They are not rules in the classic sense, they

[2]In the strict route-planning interpretation, a route is a set of x, y and z coordinates defining the path flown by an object.

are rules-of-thumb that are *usually* correct. Their primary purpose is to nudge the system away from wasteful exploration, without giving explicit guidelines on how to find the best maneuvers. In MAVERICK, the heuristic implementations are basically maneuver transforms.

3.5.3 General Operation The high level operation of MAVERICK is a fairly simple loop. Maneuvers are treated as LISP objects, and a root maneuver consisting of the NULL maneuver is generated. In effect, this means the basic straight-line path is always flown first. From this initial trial, various new maneuvers are generated, and are placed upon the agenda as maneuvers to be tested. Then the most interesting maneuver is selected from the agenda and tested; it in turn may generate new maneuvers, and the cycle repeats. Figure 3.2 illustrates this looping behavior.

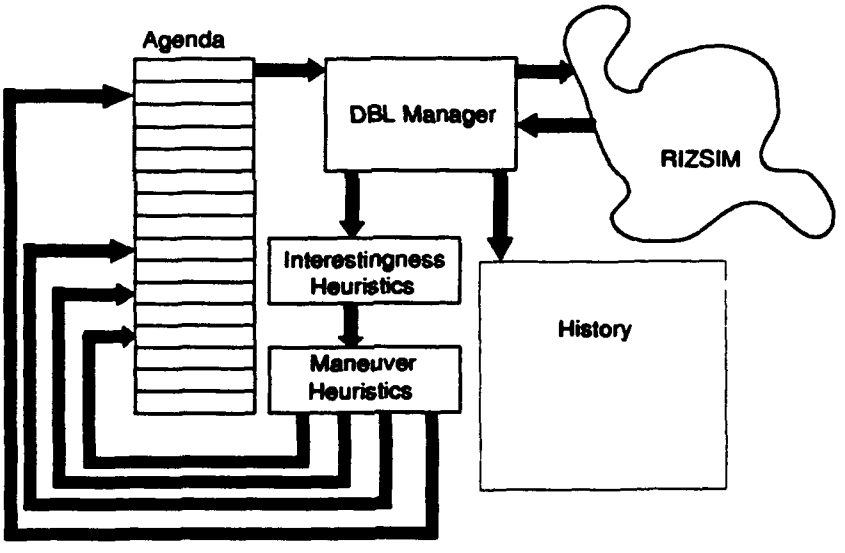


Figure 3.2. MAVERICK Flow Diagram

Figure 3.3 shows a hypothetical exploration tree "snapshot" during exploration. Maneuvers that have been tested can generate new maneuvers (dark lines in the tree), while maneuvers that have not been tested have no children in the tree (dotted lines). A good set of heuristics will limit the "bushiness" of the exploration tree by reducing the number of low quality maneuvers considered on each branch.

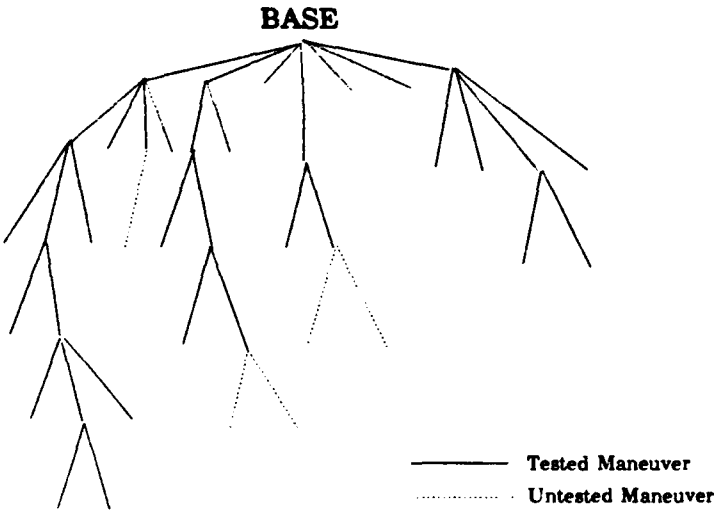


Figure 3.3. Theoretical MAVERICK Exploration Tree

3.6 Summary

The methodology used in the design of this thesis involved three main steps: choice of domain, choice of learning method, and choice of language. Route planning was chosen as a domain which was appropriate and reasonably scoped. This led to the selection of DBL as an interesting learning method to explore. Finally, LISP was chosen as the best language for a prototype system. The architecture of MAVERICK was then designed to incorporate a general purpose DBL learning structure which was adapted to work with RIZSIM for exploration in the route planning domain.

IV. The MAVERICK Discovery-Based-Learning System

4.1 Overview

The two main components of the MAVERICK system are the Discovery-Based-Learning (DBL) shell and the AFIT Battle Simulation (RIZSIM). These components run together on a single platform and communicate through the use of data files.

4.2 RIZSIM

RIZSIM is a general-purpose battlefield simulation written as part of Capt Robert Rizza's Master's Degree thesis at AFIT in 1990 (20). The original purpose of this model was to provide a "representative" battlefield simulation that could be ported to parallel computing research at AFIT (21). As part of this objective, the simulation was written to be easy to understand and modify, making it ideally suited to a DBL application. To allow for ease of modification, many "hooks" were left to allow for later simulation improvements. For example, a line of sight function is provided as part of the simulation operation, but is basically a null function (i.e. it always returns TRUE). This makes modification easier, because the line of sight function can be inserted later with no modification to the rest of the simulation. However, this can also cause a fair amount of confusion, as it is not always clear which components of the simulation model are operational, and which are just hooks.

4.2.1 Event-Driven Simulations There are two main classes of simulations, time-driven and event-driven. In a time driven simulation, a simulation clock advances the simulation by a fixed increment, and object positions are updated based upon this increment. Thus object interactions are ultimately controlled by the length of the time increment. The simulation calculates updates on every time increment, whether anything interesting is happening in the simulation or not. If the increment is too small, then the system will suffer a performance penalty because of having to perform too many updates on objects. If the time increment is too large, then the system may miss

interactions between the objects (events), because they happen too quickly and fall between two "ticks" of the clock. In most combat environments, objects move independently for extended periods of time, have intense interaction for short periods of time, then continue to move independently. A time driven simulation may waste time during the initial stages of the simulation, then not be at a sufficiently small granularity to accurately model the actual combat.

Event simulations work under a different philosophy. In an event driven simulation, the clock is only updated when an event occurs. Typical events are such things as object entered sensor range, object collision, object reached turn point, object fired weapon, etc. Objects move in discrete jumps until the next event occurs, whereupon their positions are updated. In contrast to a time driven simulation, the time scale will be very large at the beginning of the simulation, and objects will almost instantaneously jump to the first sensor contact event. At this event, the time scale will effectively increase in granularity to whatever degree is needed to handle the interactions among the simulation objects. This makes the event driven approach ideally suited for combat modeling.

4.2.2 RIZSIM Objects The objects in RIZSIM are modeled as generic moving objects. All objects have an initial route to travel and an initial position, as well as a 3-dimensional initial velocity vector. In addition, every object may have sensors of a certain range, and missiles as armaments. The list of attributes for an object is much more extensive, but the remaining attributes are not currently used by the simulation. However, the implemented attributes can be used to model objects such as trucks, tanks, aircraft, buildings, and SAM's. Soderholm (21) states that stationary objects cannot be modeled because objects that reach their last route point lose the capability to sense. While this is true, stationary objects can be effectively modeled in a different way. An object that moves along a very short path at an extremely slow rate is effectively stationary, within the accuracy boundaries of the simulation. In MAVERICK, SAM's are objects that move one simulation unit, at a velocity of .00001 units/second. This ensures they are always moving, thus always able to sense objects. During the course of the simulation, they do not move far enough to

affect accuracy, and they do not complete their "route" until well after the combat of the simulation is completed. Because the simulation is event driven, no performance penalty results. After the aircraft completes its route, the SAM's immediately jump to the completed state.

4.2.2.1 Missile Behavior Because the simulation objects must start out with pre-determined routes, it is difficult to accurately model the pursuit behavior of a missile. In RIZSIM, missiles are scheduled with three route-points: the missile's current position, the target's current position. and the target's next route-point. This is a very minimalistic approximation of a missile's behavior. To evade the missile, the target simply has to change its velocity vector before the missile reaches it.

4.2.3 RIZSIM Operation The most frequent operation of RIZSIM is the sensor_check function. The next route-points of all objects are checked to see if a sensor contact with any other object will result before that route-point is reached. If not, then object's position is updated. If a sensor contact is indicated, the the object may perform some action upon reaching the sensor contact point. Reaching turn-points and entering sensor contact are the two primary events that occur in RIZSIM, and they are scheduled into a queue sequentially in time. It is important to note that all objects move along their pre-determined route until they have a sensor contact. This is the only event that will cause an object to possibly deviate from its course. The comprehensive list of events implemented in RIZSIM is as follows:

- Reached Turnpoint
- Entered Sensor Range
- Made Sensor Contact (converse of above)
- Ordinance Released
- Ordinance Reached Target

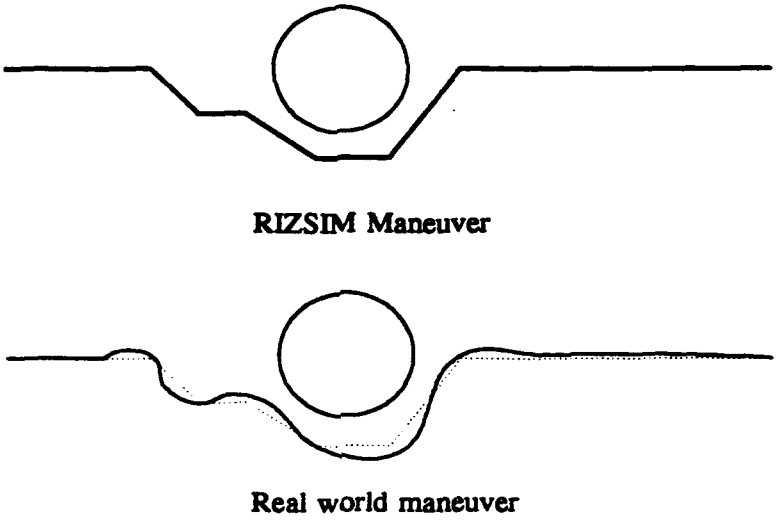


Figure 4.1. RIZSIM vs Real Routes

Object velocity vectors are determined by the relative vector between the current route-point and the next. Because of this, objects in the simulation make what is essentially a "flat" turn, and change velocity (heading) instantaneously. An object's route consists of a series of discrete line segments. While this may seem unrealistic, within limits it is a reasonable approximation of a real aircraft's behavior. Figure 4.1 shows a comparison of RIZSIM maneuvers versus the equivalent real maneuver.

4.3 RIZSIM modifications

There were three main areas of RIZSIM modifications:

- Input function enhancement
- Operator evaluation function enhancement
- Output enhancement

All modifications were implemented with prime concern for modularity. No new functions were introduced; the old RIZSIM functions were simply replaced with the new modified versions.

4.3.1 Input function The function **read_datfile** in RIZSIM is used to read the attributes of the simulation objects from a data file. This data file is essentially a scenario file because it contains the routes of all the objects in the simulation, and this determines what will happen in a given scenario. The format of the data file is a very long string of numbers, followed by a carriage-return, for each object. An object with a full set of attributes will have over forty-six fields in its one line of the data file. To further complicate matters, the fields in the line are not fixed. For example, the num_routepoints field determines how many of the next groups of three fields are routepoints. Thus, the position of the fields after this point is dependent upon the number of routepoints. This makes quick modification of scenario files exceedingly difficult.

The best solution to this problem would be to write a sophisticated front-end interface to assist the user with the structure of the data files, provide error checking, etc. It was felt that building a user interface was beyond the scope of this effort. However, as a compromise, the data file was made much more "friendly." This modification simply consisted of adding labels to the fields in the data file, and changing the structure so that only one field is used per line. It is important to note that the data must still be in the original order required by the original function. The new input function ignores the labels and reads the data sequentially. The labels greatly speed the process of scenario modification because the desired value can be accessed immediately and painlessly.

No new attributes were added to the simulation nor were any deleted. The new function still uses the original set of attributes. Thus, later improvements to the input function can easily be accommodated, or the old function can be used, if desired.

4.3.2 Operator Evaluation Enhancement The **operator_evaluation** function in RIZSIM is essentially a primitive expert system which tells the "operator" of any object what to do when a sensor contact is encountered. It is interesting to note that in object-oriented fashion, no distinction is made between the operator of a missile, a plane, a truck, or any RIZSIM object. In its operation:

- Missiles do nothing.
- Allies evade each other if they are on a collision course, otherwise they ignore each other.
- Enemies are attacked if the current object has been detected, or if they are on the target list.
- Otherwise, enemies are evaded.

This is a reasonable system, given the scope of the simulation. However, it provides little utility for a learning system, so it was eliminated for the purposes of this thesis. In fact, the entire **operator_evaluation** function was eliminated, except for the outer skeleton.

For the purposes of this thesis, the only object which uses the **operator_evaluation** function is the object with **object_id** = 1. By definition, this is the "learning aircraft", the central element of the learning system. All other objects call the **operator_evaluation** function as part of the normal operation of RIZSIM, but they do nothing as a result of calling it.

4.3.2.1 Maneuver Representation For the new **operator_evaluation** function, I needed a method to input a maneuver to the "operator", which would be performed upon contact with a threat object. I chose to represent a maneuver as a set of turn-duration pairs. The maneuvers are strictly two-dimensional, but this is logical when considering the two-dimensionality of the RIZSIM sensors. Modeling a maneuver in three dimensions would provide no utility, but would add complexity. The third dimension could be easily added, however. By definition, positive turns are to the left, and negative turns are to the right. For example, the maneuver (10 15 -20 20) represents a ten degree turn to the left for fifteen seconds, followed by a twenty degree turn to the right for twenty seconds. Implicit in the operation of the simulation is the return of the aircraft to the next route-point after a maneuver is completed. Null maneuvers are also defined, simply by convention, as "(1 0)". This could be interpreted as a one degree turn for zero seconds, which would *act* like a null maneuver, but this is explicitly recognized as a null maneuver by the **operator_evaluation** function for reasons to be explained later.

4.3.2.2 Maneuver Conversion In the new **operator_evaluation** function, the maneuver that has been input is converted to a series of routepoints for RIZSIM. These routepoints are generated based upon the aircraft's current velocity, or velocity during the continued execution of the maneuver. One complication occurred as a result of how RIZSIM maintains a data structure of the routepoints for a given object. The routepoints are stored on a stack, making the addition of *one* route-point in the middle of an object's flight quite easy by simply pushing it onto the stack to become the next route-point. However, to add *multiple* routepoints to an object, they must be pushed in reverse order (last route-point pushed first). Because routepoints are generated sequentially as a maneuver is processed, this required pushing them onto a temporary stack, so they could be popped in reverse order onto the object's route-point stack.

4.3.2.3 Multiple Maneuver Representation Early in the development of MAVERICK, I realized learning would be much more interesting in a multiple object scenario. This required the ability to further enhance the **operator_evaluation** function to allow it to execute different maneuvers for different objects in the scenario. For this to occur, the particular null maneuver "(0 0)" was selected to act as a "spacer" between different maneuvers. The "key" for which maneuver to execute is defined as the **object_id**, which is simply an increasing integer number for each object in the simulation. The maneuver for each object always starts with "(0 0)." To illustrate this, the following is a typical maneuver file for RIZSIM:

0	0	Object 1 is the primary aircraft, so no maneuver is generated for it.
0	0	This is the header for object 2.
20	10	A twenty degree left turn for 10 seconds.
-20	10	A twenty degree right turn for 10 seconds.
0	0	This is the header for object 3.
0	0	This is the header for object 4. Object 3 has no maneuver associated with it.
0	5	This is a delay maneuver. It simply means "do nothing for five seconds."
-10	1	A ten degree turn for one second.

4.3.2.4 Error-checking Because this is a prototype system, no error checking is provided for the format of the maneuver file. If it is not in the specified format, unpredictable results will occur. Experience has shown that RIZSIM will typically process the file successfully, but the results of the maneuver are unpredictable.

4.3.3 Output Enhancement RIZSIM is set up to output to a generic display driver file. This file can be read by a program running on a Silicon Graphics workstation to display a three-dimensional representation of the RIZSIM run. This display is fairly slow, however, and does not provide much utility to the learning system development. The primary reason for this is because the objects in a learning scenario are typically one moving object (the primary object) and lots of stationary objects (the threat SAMs). Viewing this in three dimensions would provide minimal feedback as to the utility of any given maneuver.

4.3.3.1 Gnuplot Files To provide a quicker, more useful output "display" function, the **update_position** function in RIZSIM was patched to write to an extra file named "man.out" whenever it writes to the generic display driver file. Whenever the primary object's position is updated, RIZSIM writes the aircraft's X, Y, and Z coordinates as well as the current simulation

time at the time of the position update. This creates a file of coordinates which defines the aircraft's route of flight.

I also found it useful to have a visual representation of the range of the SAM sites in a scenario. To accomplish this, MAVERICK looks at the simulation data file and determines the location and range of all the SAM sites. It then uses this information to generate data files with names "1.sam", "2.sam", etc. up to the number of SAM's in the scenario. GNUPLOT[1] can then use these points to plot a visual representation of the SAM ranges.

It is important to note the learning component of MAVERICK does not use this information whatsoever.[2] The only reason the program looks at the scenario data file is to generate plotting points for GNUPLOT.

MAVERICK generates several output files to illustrate the progression of learning. These files can all be plotted with GNUPLOT or other similar programs. Using GNUPLOT, they can be viewed at any time during the course of learning. The following files are generated during each run.

route.history This is a cumulative listing of *all* the route-points flown during learning. When this file is plotted, it shows the total picture of all maneuvers that have been tested.

ai.out This file is generated by RIZSIM and is the single current maneuver being tested. It is overwritten by each new maneuver that is tested.

interest.history This is a cumulative list of the interest of maneuvers tested during learning.

man-time.history This is a cumulative list of the length (time) of each maneuver tested during learning.

[1] GNUPLOT is a program which plots graphs from a data file consisting of x and y coordinates. Any similar plotting program could be used.

[2] In other words, MAVERICK does not "cheat" in the learning process by reading the SAM's location and range from the data file.

radar.history This is a cumulative list of the *total* radar coverage of each maneuver tested during learning.

Hn.history A history file is created for each heuristic, showing its "worth" over the exploration.

4.4 MAVERICK Objects

MAVERICK is written entirely in Common LISP, using the Common Lisp Object System (CLOS), wherever appropriate. The structure of the problem does not map to an extensive object-oriented approach; there is no inheritance needed between objects. The primary objects in MAVERICK are the maneuver object, the agenda object, and the scoreboard object. To prevent confusion, the discussion below will use the term "scenario objects" to refer to objects encountered by MAVERICK during exploration (typically threats).

4.4.1 Maneuvers MAVERICK is built to manipulate and generate maneuvers, so the primary object is quite complex in features. Each maneuver has a unique identifier, generated by the Lisp function **gensym**. This identifier is used to reference individual objects. Maneuvers comprise the following slots:

interestingness An integer value indicating the interestingness of the maneuver.

move The actual set of turns in the maneuver. This slot is a nested list, with one move per object in the scenario.

age An integer counting variable, showing the sequence of the maneuver in the total number of maneuvers explored.

heuristics A list of which heuristics were used to form this maneuver.

level An integer indicating the tree-depth level of the maneuver.

man-time A float indicating the total trip-time for the maneuver in seconds.

total-radar-contact A float indicating the total radar contact time, in seconds, for the maneuver.

radar-contact A list of the individual scenario object radar contact times.

radar-directions A list of the direction for radar contact for each object, either "L" or "R."

exec-time A float indicating the total time required for the computer system to test the maneuver (CPU time).

child A list of the children of the maneuver.

parent The parent of the maneuver.

Virtually all of the slots are updated when the maneuver is tested, except the child slot which is updated whenever a child maneuver is spawned. The exec-time slot is not used by MAVERICK.

4.4.2 Agenda The agenda is used to control the activities of the DBL process. Originally, the intent was to have a variety of activities on the agenda in addition to the basic "test maneuver" activity. These activities would have been basic housekeeping functions such as collecting the best maneuvers, applying heuristics, and updating heuristic utility values. The exec-time slot was originally intended to provide guidance for trade-off determinations as to which activities to perform instead of testing maneuvers. However, it quickly became apparent that any housekeeping function time was completely dominated by the time it takes to test a maneuver. The entire body of housekeeping functions could be completed in a small fraction of the time needed to test a maneuver, so it made no sense to further complicate the operation by putting these activities on the agenda. In the present operation, the only activities on the agenda are test maneuver activities, and all housekeeping functions are performed after every test. However, the format of the agenda allows for future expansion to more general activities.

The format of the agenda is quite simple, comprising only two slots:

age An integer indicating the total number of activities performed from the agenda.

body An ordered list of tasks.

4.4.2.1 Agenda Tasks The tasks on the agenda are designed to be general-purpose objects to allow the execution of a variety of activities. Tasks have the following slots:

task-interestingness An integer value indicating the interestingness of this task. This determines the task's precedence on the agenda.

age An integer value indicating how long the object has been on the agenda. (not used)

to-do A symbol indicating what task is to be performed.

task-maneuver Which maneuver is to be tested, in the case of a test maneuver task. This slot could be used for any argument needed by a to-do symbol.

4.4.3 Scoreboard The scoreboard is used to give the DBL system guidance to determine when to stop exploring maneuvers for one scenario object and move on to other scenario objects. It also provides a storage location for the "scenario memory", used to allow MAVERICK to apply learned maneuvers from one scenario object to other scenario objects. The scoreboard uses the following slots:

age A list of integers indicating the number of maneuvers explored since the best maneuver for a given object has changed.

max-objects An integer indicating the total number of objects that MAVERICK has found in the simulation.

best-maneuvers A list of maneuvers indicating the best maneuver found for each scenario object.

zero-maneuvers A list of all maneuvers resulting in zero radar contact for any object.

best-radars A list of floats indicating the best radar contact time found so far for each object.

4.5 Maverick Operation

4.5.1 Hill Climbing One potential problem with many AI systems is their tendency towards "hill-climbing" behavior. Pearl (16) claims hill-climbing is the simplest and most popular search strategy among humans. The basic strategy behind hill-climbing is to use ***local knowledge*** about the space around the particular search state to guide the system towards the maximum gradient of the evaluation function.(6)

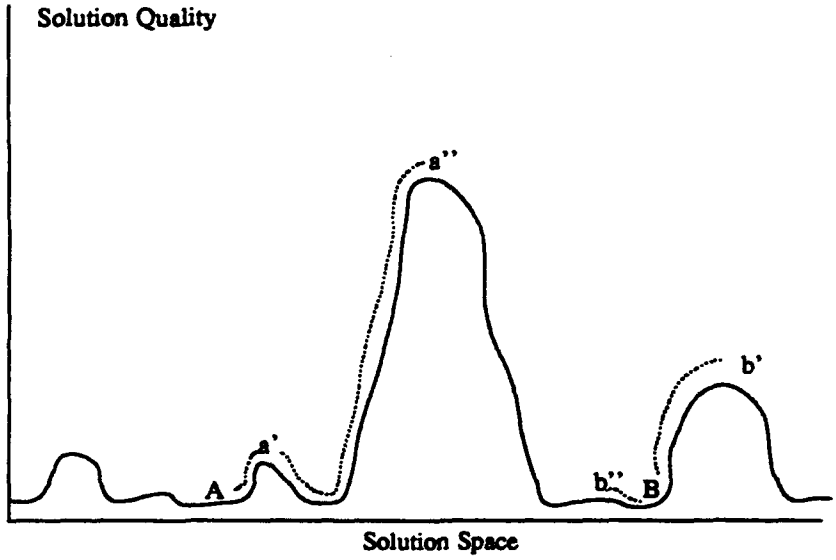


Figure 4.2. Typical Hill Climbing

This is graphically illustrated in figure 4.2. In the hypothetical search space, the good solutions are shown as smaller peaks, while the optimal solution is the center tallest peak. Initial state spaces are shown by **A** and **B**. From **A**, a hill-climbing system will take the steepest route and stop at location **a'**. In a pure hill-climbing system, the search will stop at **a'**, because there is no positive gradient from this location. This is a local maximum; the solution can only be improved by first making the solution worse, then climbing to **a"**. Similarly, state **B** could come closer to the optimal solution by proceeding towards **b"**, but would take the steeper gradient towards **b'**.

Figure 4.3 illustrates another difficulty with hill climbing. An especially difficult problem for a hill climbing system is a solution space with a sharp "peak", especially when it is surrounded by

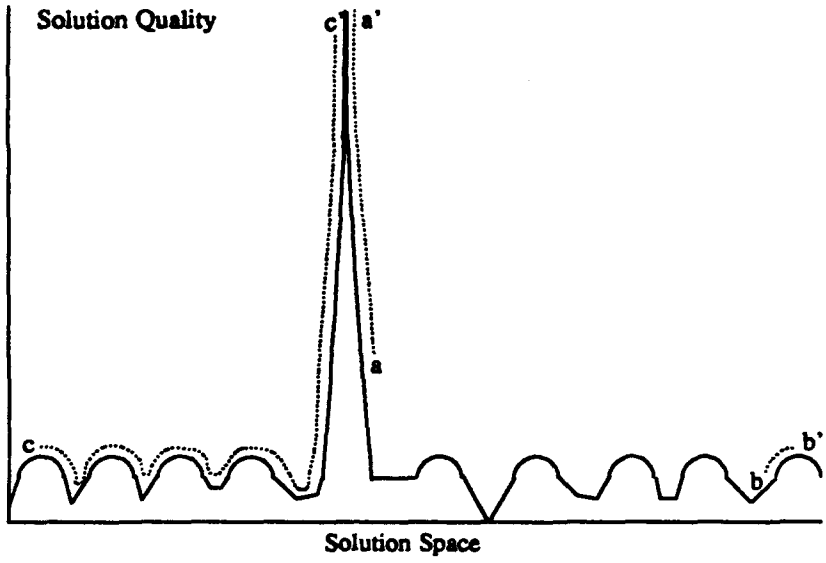


Figure 4.3. Hill Climbing with Peaks

local maximums. In this hypothetical space, state c will have to abandon several good solutions before arriving at the optimal solution at c'. If the system is lucky enough to start at state a, then it will perform well. However, the narrowness of the optimal solution makes this more unlikely.

Figure 4.4 shows my estimation of the search space for maneuvers in route planning. It is characterized by sharp peaks; a small manipulation of a good solution can quickly result in a very bad solution. There is an optimal solution that is typically in a very narrow range with moderately good solutions around it.

Figure 4.5 shows a variety of potential solutions to a single SAM site scenario. Solution A is the null maneuver. Solution B is a better solution, primarily because it has no radar contact. Note, however, that it does have a worse total trip time than maneuver A. Maneuver C is still better because it also has zero radar contact, but has less total trip time. Maneuver D is an optimal maneuver because it has the minimal trip time while maintaining zero radar contact.

In a DBL system, the search strategy is much less structured than in a hill climbing system. In DBL, the system bounces back and forth between potential solutions, based upon which solution

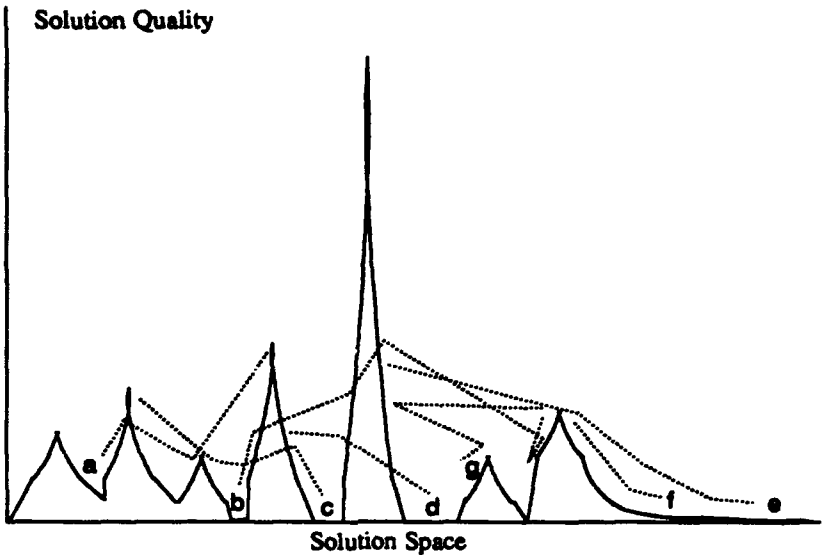


Figure 4.4. DBL Hill Climbing

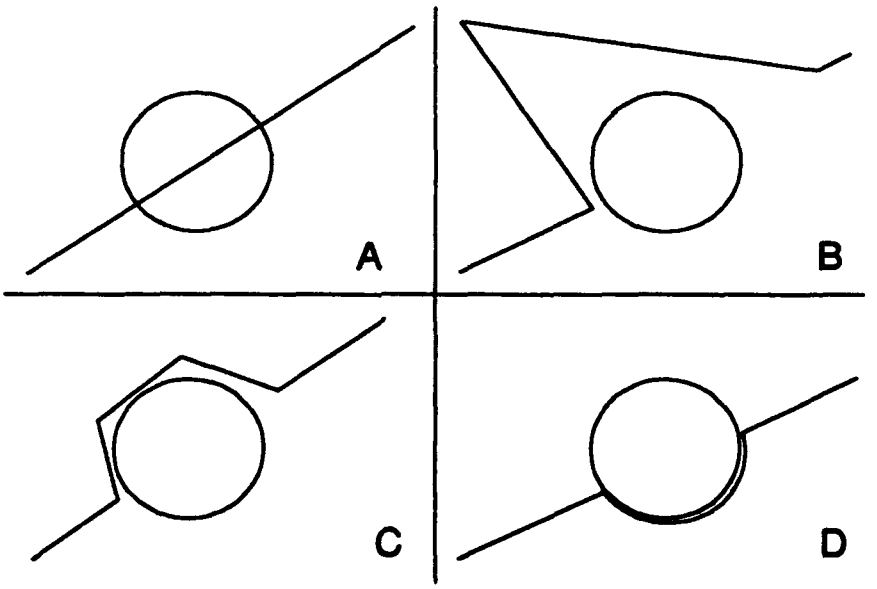


Figure 4.5. Potential Solutions

is judged to be the most "interesting" at any given time. This is illustrated in figure 4.4. The interestingness measure considers minimizing radar contact and trip time, but also considers other factors which may lead to a better solution. Also, because the system manipulates entire maneuvers, it frequently transforms a maneuver into something quite different, further reducing the hill climbing tendency. However, MAVERICK is not immune to hill climbing, Chapter 5 illustrates some of MAVERICK's hill climbing behavior.

4.5.1.1 Hill Climbing Prevention One explicit technique I used to minimize hill-climbing in MAVERICK was *maneuver aging*. The basic idea is quite simple: for a given object, every time a new "best" maneuver is discovered, set that slot in the scoreboard to zero. A hill climbing factor is used to determine when the current best maneuver has been around long enough to be considered the best move that can be found. If this factor is zero, then the system acts in a pure hill-climbing fashion, finding the very first acceptable solution for the object, then terminating. Higher hill climb factors produce better solutions until a point of diminishing returns is reached.

4.5.2 Heuristics One of the major components of MAVERICK is its set of heuristics. The heuristics are designed to provide guidance without pointing to a specific desired solution. The heuristics are divided into two groups: maneuver heuristics and interestingness heuristics. The interestingness heuristics determine the interestingness of a tested maneuver, which determines a maneuver's precedence in future exploration. The maneuver heuristics are applied to tested maneuvers to suggest further maneuvers to explore.

In the following discussion, pseudocode will be used to describe the heuristics. The LISP form may be unclear without an understanding of the total operation of the program.

4.5.2.1 Interestingness Heuristics The interestingness heuristics are quite simple. Each one is of the format: if <condition>, then add bonus. The bonus numbers are a human estimation of the relative interestingness of the different factors. No negative numbers are used; a maneuver

that is not interesting simply gets no bonus. The following heuristics were used, along with the "default" bonus weighting.

1. If a maneuver results in zero radar contact (1)
2. If maneuver's radar coverage is improving (50)
3. If a maneuver's trip time is improving (10)
4. If a maneuver's radar coverage changes from anything to zero or vice-versa (30)
5. If a maneuver is the best so far (20)
6. If a maneuver improves both radar coverage and trip time (30)

The heuristics are fairly self-explanatory, with the possible exception of 4. In this case, a maneuver is likely to be very close to an optimal solution, because it is just at the threshold. This tends to concentrate effort at this point.

4.5.2.2 Maneuver Heuristics The maneuver heuristics are somewhat more complicated than the interestingness heuristics. The basic form is: if <condition> then do <transform>. The transforms are various manipulations that change the maneuver's form, hopefully to improve it. The difficulty in designing heuristics arises from determining how much information is needed for guidance, without leading to a specific desired conclusion.

Due to the nature of the maneuver heuristics, English-type descriptions are more informative than pseudocode.

1. If no maneuver history exists for the current object, then turn in a continuous number of directions (-60 degrees to + 60 degree) to attempt to establish a starting population of maneuvers. Start the set of turns based upon the direction of the radar contact. Also, use maneuvers generated from other objects, if available.

2. If a maneuver has zero radar contact, then try scaling it down (to reduce trip time).

3. If a maneuver has been scaled down repeatedly, and still has zero radar contact, then try scaling it down by a greater magnitude.

4. If a maneuver has positive radar contact, then try scaling it up (in effect, turn farther away from it).

5. If a maneuver has zero radar contact, then try short-cuts across legs of the maneuver (to also reduce trip time).

6. If a (learned) maneuver is intended for an opposite direction radar contact, then try a mirror image of it.

7. If a maneuver has zero radar contact, then try starting it later.

8. If a maneuver has positive radar contact, then try starting it earlier.

The nature of the conditions reflect the rule-of-thumb nature of the heuristics. For example, if heuristic 2 is applied to an optimal, or near-optimal solution, it will probably increase the radar contact to a near maximum.[3] However, *most* of the time this heuristic will work.

4.5.3 Heuristic Ordering Ideally, the order of the heuristics should have no effect on the discovery process. However, because of the linear nature of the agenda, the order of the maneuver heuristic application does have some effect. The reason for this is that the maneuver heuristics are used to spawn new maneuvers. All the new, untested maneuvers spawned from a given maneuver will have the same interestingness (their parent's). This means they all have the same precedence on the agenda. However, they will have to be placed upon the agenda in *some* order, and this order is typically the order in which they were generated, which is also the order in which the heuristics are applied. This gives an "unfair" advantage to heuristics which are applied earlier, because the

[3] A near-optimal maneuver will just skirt the outside of the radar contact zone. Scaling it down will cause it to just skirt the *inside* of the radar contact zone.

maneuvers generated from the early heuristics are tested first, which will in turn generate more maneuvers from the early heuristics, and so on. The maneuvers generated by the later heuristics will not be tested until later.

To partially get around this problem, and to potentially increase performance, I created a scheme to take advantage of the performance history of the heuristics. Whenever a maneuver is tested, the value of the heuristic which generated that maneuver is increased or decreased by the change in interestingness for that maneuver. Also, because some heuristic applications work better in tandem, the heuristic which preceeded the current heuristic is also given the positive or negative bonus by a smaller amount. Therefore, heuristics which produce good results get higher values. Note that these values change over time; heuristics which are valuable early in exploration may not be valuable later. These values are used to control the order in which the heuristics are applied. All heuristics are still applied, but the heuristics which were the most recently useful get precedence.

It is important to note, however, that this technique does not totally eliminate heuristic order effects, it just minimizes them. The reason they are not eliminated is because there is some time needed to establish a history. The order of the heuristic application will have an effect until a history is established.

4.5.4 MAVERICK Learning & Memory The discovery process itself is a form of learning. In addition, MAVERICK utilizes learned information in several other aspects of its behavior. The first of these is the previously mentioned heuristic history. By using this history, MAVERICK determines which heuristics are working the best at the current time, and keeps using these as long as they continue to work. A second method is *scenario memory*. Scenario memory is the application of learned maneuvers to other objects in a given scenario, with the hope that the learned maneuvers will provide a short-cut to a solution, compared to starting from scratch again. A final form of memory is long-term memory. Under this technique, MAVERICK writes to a file the optimal maneuvers learned for objects, along with some characteristic information about that

object. Then, if objects with similar characteristics are found in later scenarios, MAVERICK can attempt to apply the learned maneuvers. Chapter 5 presents an analysis of the effectiveness of the various types of learning and memory.

4.6 Summary

MAVERICK was designed to work with RIZSIM, a general purpose battlefield simulation. RIZSIM was modified slightly to ease its interface with MAVERICK. The design of MAVERICK included heuristics to guide exploration, a method of manipulating and generating maneuvers, and an interface to RIZSIM for MAVERICK to test new maneuvers. The design also incorporated techniques to prevent negative hill climbing behavior and various learning techniques to improve the performance of MAVERICK over time.

V. Results & Issues

During the analysis of the performance of MAVERICK, four main operational features were varied to provide for characterization of the program. These features were as follows:

Heuristic Sorting This feature provides dynamic sorting of maneuver heuristic application during program execution. This feature is either on or off.

Hill Climb Factor This feature determines the depth of exploration MAVERICK uses to offset the adverse effects of hill climbing behavior. This feature can be set from zero (pure hill climbing) to any integer.

Scenario Memory This feature allows the use of maneuvers learned during exploration of early scenario objects to be applied to later scenario objects. This feature is either on or off.

Long-Term Memory This feature allows MAVERICK to use maneuvers learned from other scenarios on the current scenario. This feature is either on or off.

This analysis will present the basic scenario used for the majority of the tests, followed by examples of the various history files used to characterize performance on these tests. Then the effects of the above mentioned operational features of MAVERICK will each be tested to determine their effects on MAVERICK's operation. Then, some interesting fault-tolerant aspects of maneuver heuristic performance will be discussed, followed by cases where the performance of MAVERICK starts to break down. The conditions of each test are given in their respective sections. Finally, some qualitative issues of DBL development will be discussed.

During the discussion of this analysis, several terms will be used which may have ambiguous meanings. To prevent confusion, the following definitions will apply:

Radar Contact This is the amount of time the aircraft spends in the radar zone of a SAM. MAVERICK tries to minimize this value.

Trip Time This is the total time required for the aircraft to travel from the start point to the finish point, including all deviations required for threat objects. MAVERICK tries to minimize this value.

Move This is the term used to describe a maneuver component that is used for a particular scenario object. A maneuver is composed of a sequence of moves, one move per scenario object encountered.

Best Maneuver This is the maneuver with zero radar contact and the minimum trip time of all explored maneuvers. Obviously, it changes during the course of exploration. A best move can also exist for a particular object.

5.1 Test Scenario

The test scenario chosen for the majority of the analysis is shown in figure 5.1. In this scenario, the aircraft has three route-points: (0,0,0), (60000,60000,10000), and (70000,70000,10000). In the base case, the aircraft simply flies a straight line and encounters two SAM sites.

This scenario was chosen because it exercises several of MAVERICK's capabilities. First, there are two objects, so scenario memory and multi-object learning can be tested. Second, the amount of radar contact is significantly different between the two objects; and the direction of the radar contact (right or left) is also different. This provides for a demonstration of the differences in maneuvers learned for each object. Finally, the scenario is simple enough to allow numerous test runs without taking an extensive amount of time.

The arrows on the figure show the points at which the aircraft has detected the SAM sites; the other points on the diagonal line represent route-points for the aircraft. The circles around the SAM sites represent the detection range of the SAM site, with the points at the center of the circle representing the actual location of the site. Maneuvers are initiated at the point at which

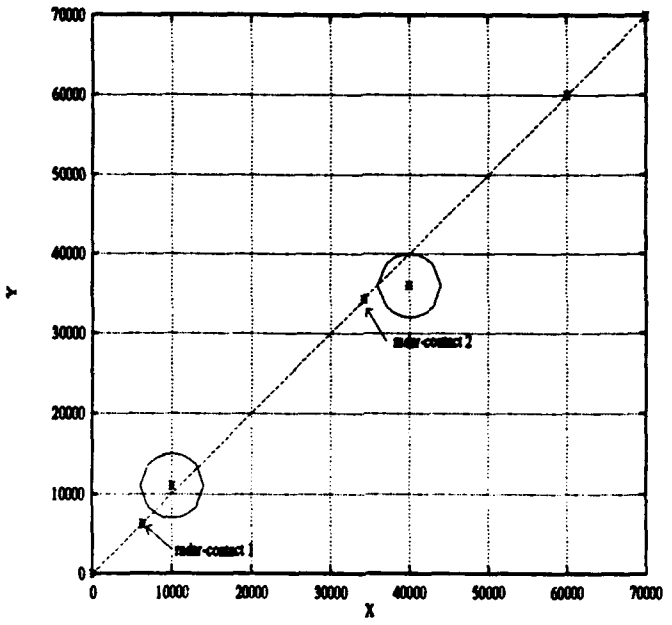


Figure 5.1. Base Case Scenario

the aircraft detects the SAM site, although in some maneuvers the initiation of the maneuver is delayed from the initial contact.

5.2 History Files

The history files produced by MAVERICK provide significant insight into the operation of the DBL process. The following history files were produced during a test run with scenario memory on, long-term memory off, sorted heuristics and a hill climb factor of twenty. The history files are presented to give a basic overview of MAVERICK's exploration process; many of these files are used to present interesting aspects of MAVERICK's behavior in later analyses.

5.2.1 Route History Figure 5.3 shows the cumulative route exploration history for a typical run. The best maneuver found for the base case is shown in figure B.2. Figure 5.3 shows MAVERICK spent very little time exploring maneuvers that have positive radar contact, while maneuvers which have zero contact are explored much more thoroughly. Numerous maneuvers which are ob-

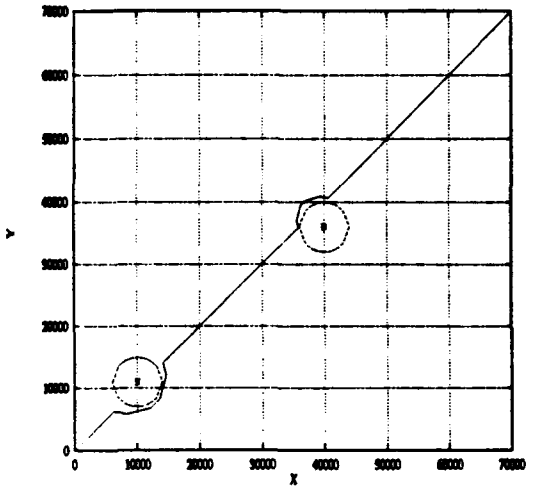


Figure 5.2. Base Case Best Maneuver

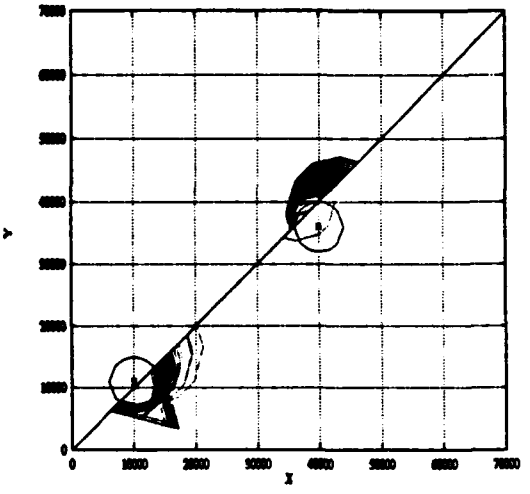


Figure 5.3. Route History

viously inferior in terms of trip time are apparent in this figure. Some of these are the result of exploration that occurred after the best maneuver had been found.[1] Others are simply stumbled upon by MAVERICK, then improved as much as possible.

5.2.2 Level History Figure 5.4 shows the level, or "tree depth," of the tested maneuvers during the run. This figure highlights the fact that MAVERICK frequently reached "dead ends" and had to back-track to previous steps in the exploration. However, it continually generated maneuvers that are acceptable and rarely had to back-track more than a few levels. In my tests, it never needed to back-track to the root level.

5.2.3 Interest History Figure 5.5 shows the interestingness of the tested maneuvers. Whenever the slope of this graph is non-positive, MAVERICK has reached the limit of the improvement of a sequence of maneuvers. At this point, MAVERICK has to return to a previously less interesting predecessor of the maneuver.

[1]MAVERICK does not accept a maneuver as the "best" until it reaches the hill climb-factor limit

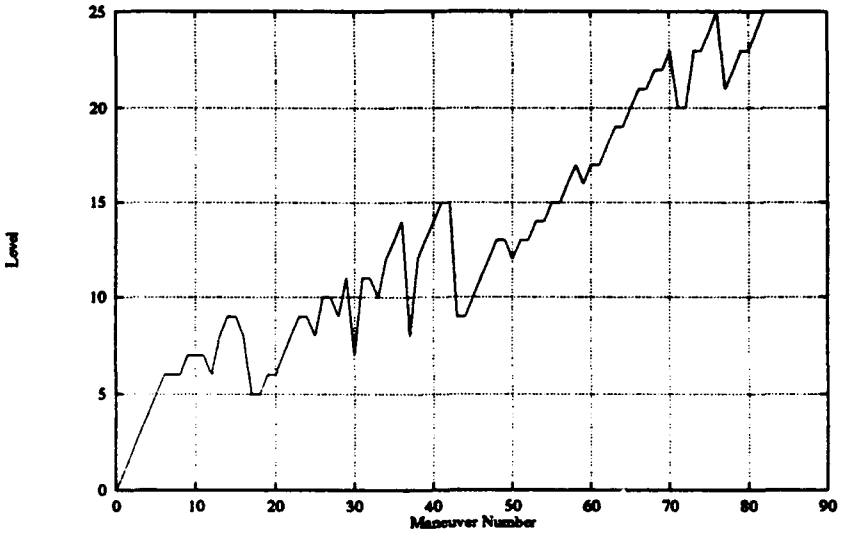


Figure 5.4. Level History

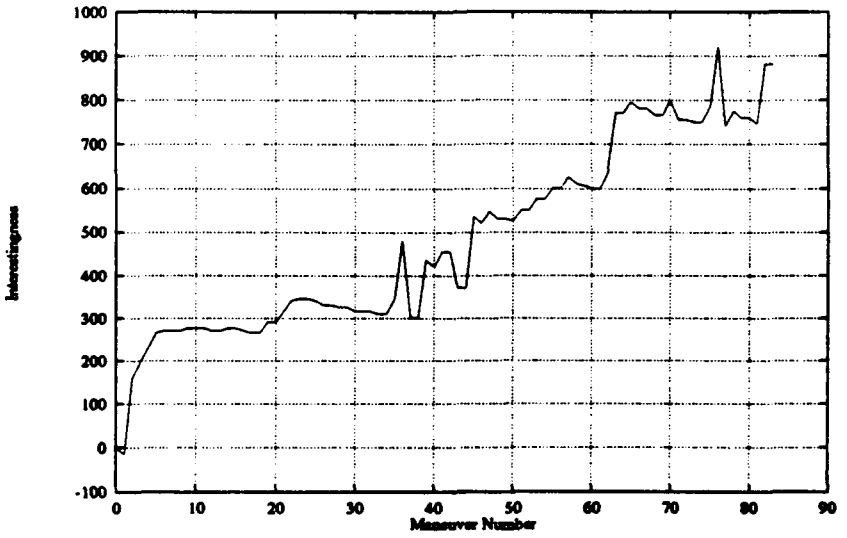


Figure 5.5. Interestingness History

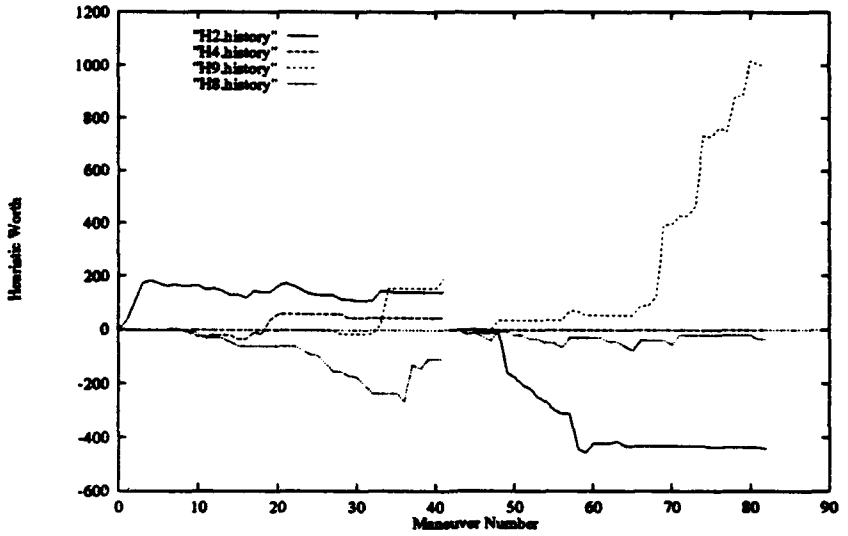


Figure 5.6. Heuristic History

5.2.4 Heuristic History Figure 5.6 shows the history of the heuristic worth values during exploration. These four heuristics were chosen from the full set of heuristics to demonstrate particularly interesting behavior, and to prevent clutter. The heuristic worth values are used to control the process of sorting the heuristic application order. When the graph has a positive slope, the heuristic has improved a maneuver; a negative slope means the heuristic has made the maneuver worse. The break in the graph is caused by the transition from exploring object 1 to exploring object 2. The heuristic worth values are reset at this point.

This graph highlights several interesting aspects of heuristic performance. First, the H2 heuristic (used to scale down maneuvers) is very useful during exploration of the first object but quite detrimental during exploration of the second object. The H8 heuristic (used to delay the start of a maneuver) only helps in a few select instances but is generally not used for positive benefit. The H9 heuristic is the counterpart to H8; it is used to remove delays from maneuvers. This heuristic is significantly beneficial to the exploration of object 2 because object 2 inherits a large set of maneuvers with delays from object 1. H9 removes some of these delays, making the maneuver much more useful for object 2. The H4 heuristic is used to provide shortcuts across the

	Sort, Trip Time	No Sort, Trip Time	Sort, # Maneuvers	No Sort, # Maneuvers
Std Dev	2.29	3.04	13.96	11.15
Mean	367.12	367.78	80.25	75.26
Median	366.45	366.45	78	78
Min	365.38	365.50	54	54
Max	372.81	375.35	101	90

Table 5.1. Heuristic Sorting Statistics

legs of a maneuver. This heuristic is quite useful during the fresh maneuver development of object 1. However, because object 2 has such a large set of maneuvers inherited from object 1, H4 does not apply to these refined maneuvers, and it has a value of zero for the remainder of exploration.

5.3 Heuristic Sorting

To test the actual performance of heuristic sorting, MAVERICK was run on the test scenario for 100 runs, using a randomly ordered set of heuristics for each run. During these tests scenario memory was active, and the hill climb factor was twenty. With each random ordered set of heuristics, the system was tested with heuristic sorting and without.

Table 5.1 shows the associated statistics for these tests. Using trip time as a benchmark, the sorted performance is very similar to the non-sorted performance. Using sorted heuristics, the standard deviation and worst-case maneuver are noticeably better, but not by a substantial amount. The median solutions found under both tests are very close to the minimum value.

Using the total number of maneuvers explored as a benchmark, the performance of the sorted heuristics is actually slightly worse. It is important to note, however, that the standard deviation for the total number of maneuvers explored is noticeably higher than the standard deviation for trip time. This shows the heuristic order is a more important factor in the *efficiency* of the DBL process than in the quality of solutions found. It also shows the quality of the solutions found are generally immune to the heuristic order.

The data presented suggests the heuristic ordering technique is not particularly effective in improving the performance of the MAVERICK. It also demonstrates that changing the order of heuristic application during exploration is probably not necessary. The system seems to perform similarly with or without heuristic ordering. However, the most significant aspect of the ordering is that it removes human intervention from the process. Looking at the minimum and maximum maneuvers explored, there are obviously some heuristic orders which work quite well, and others which do not work well. A human designer could estimate the best order in which to apply the heuristics, but this would constrain the system to his human bias and possibly hamper the system's performance in future scenarios which were different from what the designer was considering. By allowing the system to order its own heuristics, human bias is completely removed, and the performance of the *system* can be separated from the programmed performance of the human designer.

The test runs to gather this data took an extensive amount of time, but they suggest an alternative method of determining heuristic order. Some randomly generated heuristic orders generated extremely quick solutions; the minimums of 54 represent only 14 maneuvers explored to find the best solutions for both objects (the other 40 maneuvers come from the hill climb factor exploration). Given enough time, the system could try random orderings of heuristics to find which orders work best on a given scenario. Then, also given enough time, the system could try these heuristic orderings on a sample set of scenarios to try to find the best order which works across the broadest set of scenarios. The system would then be biased to the sample set of scenarios, but if the sample was a representative sample, this "learned" order could prove effective.

5.4 Hill Climb Factor Analysis

The hill climb factor is used by MAVERICK to provide some indication of when it has examined a sufficient number of maneuvers for a given scenario object. The factor determines how "old" a solution must be before it is accepted as the best solution that can be found. In effect,

the hill climb factor determines how far down the "tree" of possible maneuvers MAVERICK will travel. The first test involved varying the hill climb factor from zero to twenty-five, while using scenario memory, but no long-term memory. The results of this test are shown in figure 5.7. The two lines show the performance of MAVERICK with sorted heuristics and without to illustrate the difference in performance between these two methods. The y-axis shows the trip time of the best maneuver; shorter times are better solutions.

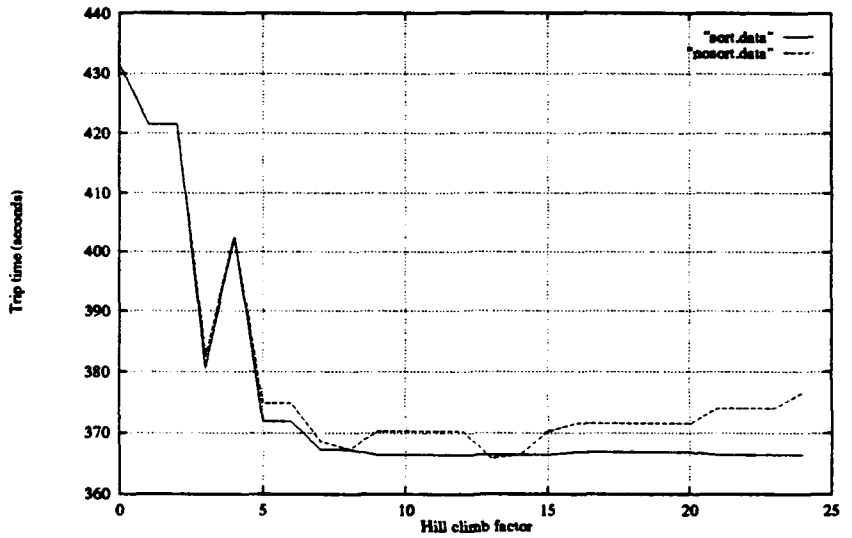


Figure 5.7. Hill Climb Factor Effects, Scenario Memory

The results are somewhat counter-intuitive. Theoretically, as the system looks deeper into the search space, better or equal solutions should be found. Solution quality should always improve or be equal: it should never decrease. The reason it should never decrease is because the system could always use the solution found at a shallower level if it cannot find a better solution at a deeper level. Thus, the spike at four and the gradual upturn of the unsorted case presented an unusual behavior.

The cause of the spike is the method in which the scenario memory operates. Scenario memory simply applies the zero-contact maneuvers learned from early objects to later objects. Since the system has not explored the later objects, there is no information it can use to determine which

learned maneuvers are the most applicable. However, the system does use the heuristic that later maneuvers are probably more refined than early maneuvers. Therefore, the learned maneuvers are applied in the *reverse* order from which they were discovered.

The intent of scenario memory is to provide a richer population of maneuvers for application to later scenario objects. This can cause problems when one of the learned maneuvers works without modification for later scenario objects. In this case, MAVERICK can start on top of a small "hill", with no way to go but down. With a small hill climb factor, MAVERICK will try a few things to improve the maneuver, then it will reach the hill climb factor limit and give up on improving that solution. With a larger factor, MAVERICK will try a few things to improve the maneuver, then move on to exploring other maneuvers before it reaches the hill climb factor limit. Figures 5.8 — 5.13 present a more detailed examination of this process. The plots on the left are the route history files for hill climb factors of 3, 4 and 5. These show the cumulative routes explored. The plots on the right show the best maneuver found for each of the cases. The maneuver in figure 5.11 is obviously worse than the maneuver in figure 5.9, which was discovered at a shallower level.

The reason for this is apparent in the route history plots for these two cases. In figure 5.8, the second object starts exploration with the trapezoidal-shaped maneuvers learned from object 1. In figure 5.10, however, the system has learned one extra maneuver, which is tried on object 2 first. This maneuver is applied first because of the heuristic that later maneuvers are more refined, and thus more useful (which is wrong in this case). This maneuver is a triangle-shaped maneuver shown around object 2 in figure 5.10. This maneuver works immediately for object 2, and MAVERICK never has enough time to move on to other maneuvers. In figure 5.12, the system has learned yet another maneuver which is applied more successfully to object 2, resulting in the better solution in figure 5.13.

Figure 5.7 also shows a decrease in solution quality of the unsorted-heuristic case at hill climb factors of eight and thirteen. After thirteen, the solution quality continues to degrade as

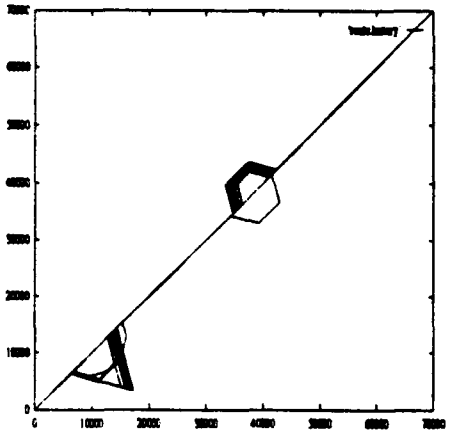


Figure 5.8. Hill Climb Factor = 3

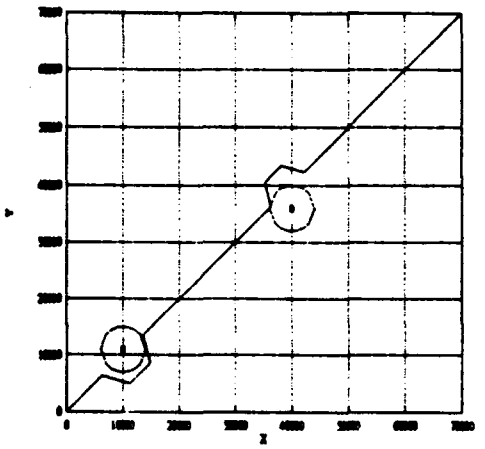


Figure 5.9. Hill Climb Factor = 3

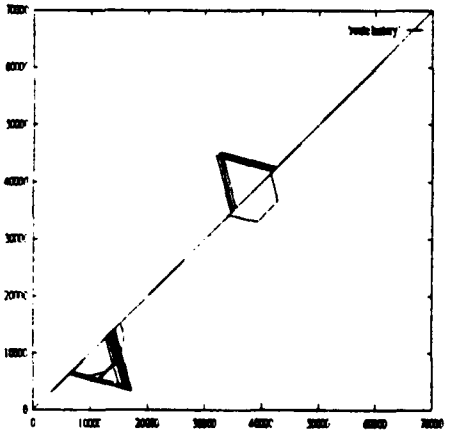


Figure 5.10. Hill Climb Factor = 4

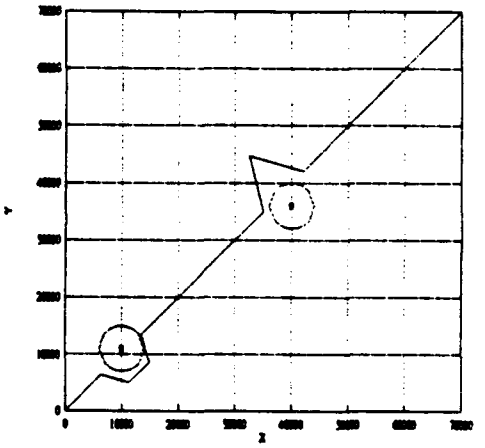


Figure 5.11. Hill Climb Factor = 4

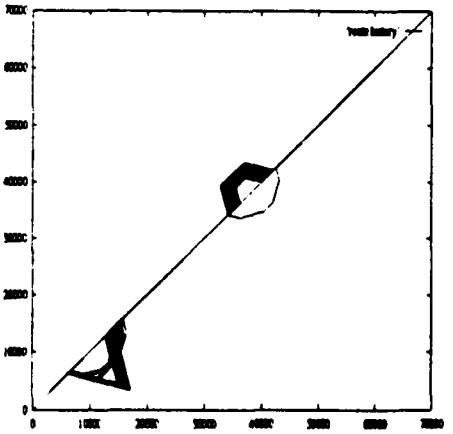


Figure 5.12. Hill Climb Factor = 5

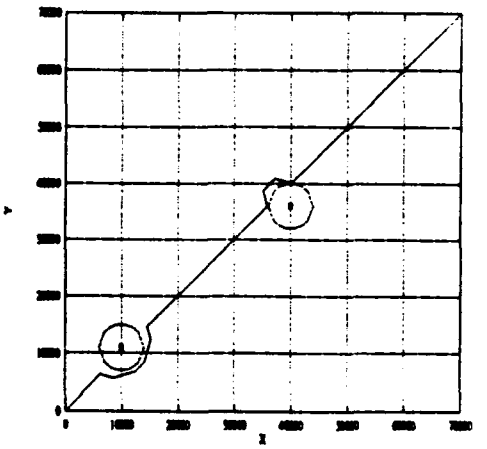


Figure 5.13. Hill Climb Factor = 5

MAVERICK looks deeper. In the sorted-heuristic case, the solution quality levels out. The evidence suggests that the cause of the degradation is also scenario memory. As MAVERICK looks deeper, it finds more potential solutions to pass on to other objects. Eventually these solutions "clutter" the search space, providing numerous "hills" that MAVERICK must eliminate to find better solutions. This effect is much less pronounced when heuristic sorting is in effect, although a very slight degradation can be seen at a hill climb factor of 15.

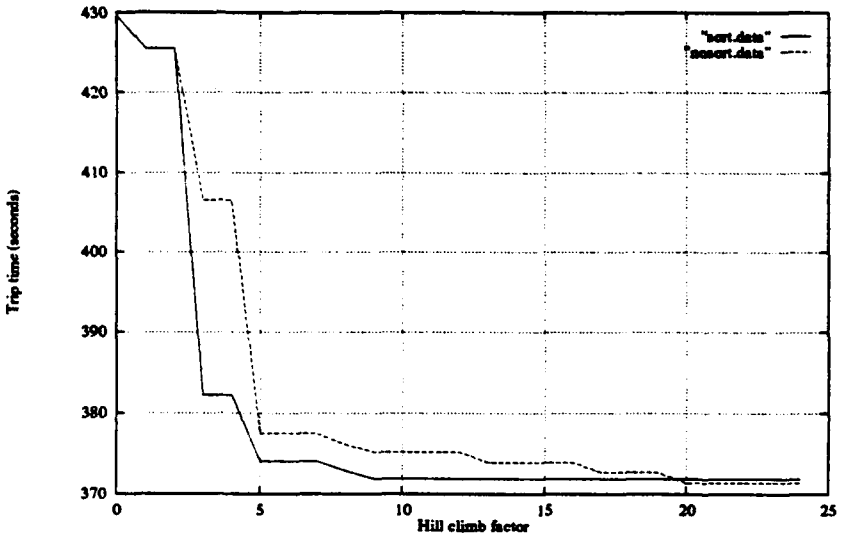


Figure 5.14. Hill Climb Factor Effects, No Scenario Memory

Figures 5.8 – 5.13 showed the spike at a hill climb factor of four was caused by an inappropriate application of scenario memory. I hypothesized that the other smaller increases were also a result of the same effect. To test this hypothesis, the same test was run with scenario memory turned off. This would eliminate all scenario memory effects because the system would start from scratch on each scenario object. Figure 5.14 shows the result of this test.

This performance is more intuitive; the solution quality continually increases, then eventually levels out. The sorted-heuristic quality increases at a faster rate, but eventually both methods level out to essentially the same quality solutions. This provides more evidence to support the above hypothesis. It is important to note, however, that the overall performance of the system suffers in

this test. All of the best solutions found without scenario memory are of worse quality than the best solutions found using scenario memory.

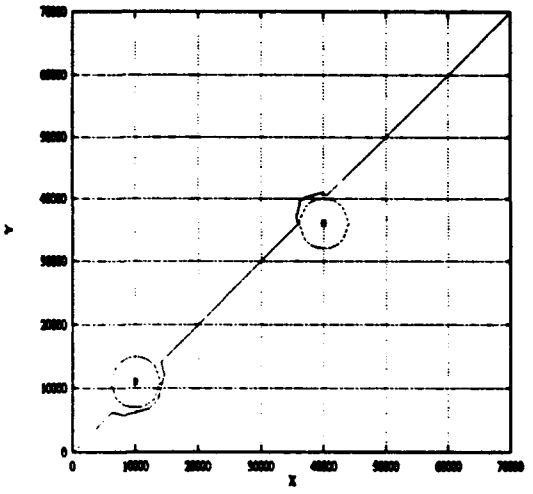


Figure 5.15. Best Maneuver, Scenario Memory

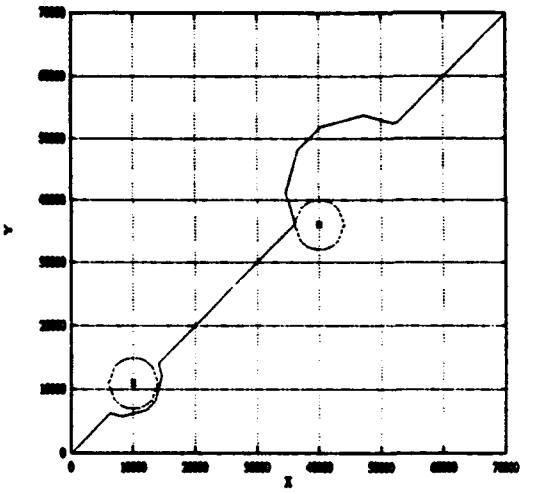


Figure 5.16. Best Maneuver, No Scenario Memory

Figures 5.15 and 5.16 illustrate the difference in solution quality between using scenario memory and not using it. These routes were both generated with a hill climb factor of twenty and sorted heuristics. Note that in this test, scenario memory was required to find a good solution.

5.5 Long-Term Memory

The operation of the long-term memory is fairly simple; when optimal solutions are found for an object, these solutions, along with the straight-line radar contact for the object are written to a file. When objects are encountered in new scenarios, this file is checked to see if any remembered solutions are available to apply to the current scenario. Currently, straight-line radar coverage[2] is the only characteristic saved in long-term memory. In a more sophisticated system, more features would be remembered such as radar frequency, and radar intensity. The current simulation does not provide this information.

[2]The amount of radar coverage when no maneuver is used

The first test run to demonstrate long-term memory involved first running the base case scenario, then "remembering" the maneuvers learned from this scenario. To validate the basic effectiveness of long-term memory, the same scenario was tested again, but with long-term memory active so it could take advantage of the learned maneuvers.

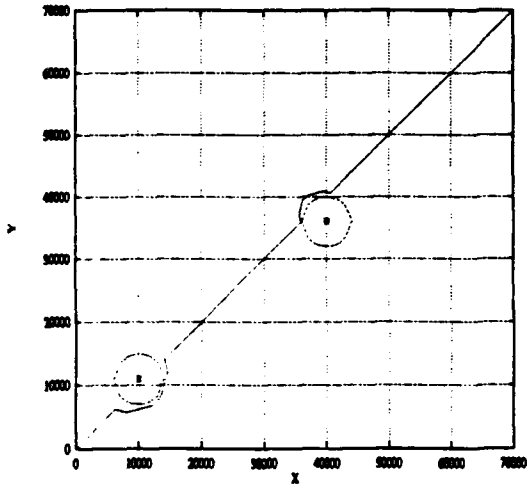


Figure 5.17. Best Maneuver, Long-Term Memory Active

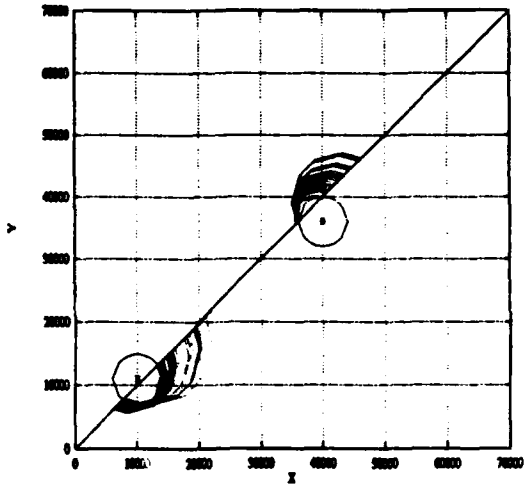


Figure 5.18. Route History, Long-Term Memory Active

Figures 5.17 and 5.18 show the maneuver solution and route history plots when long-term memory is active, and a remembered solution is utilized As expected, the maneuver solution is identical to the base case in Figure 5.15. The route history, however, shows that significantly fewer maneuvers were explored compared to the base case in Figure 5.3. As expected, MAVERICK used the remembered maneuver, then only explored twenty more maneuvers after testing this remembered maneuver (because of the hill climb factor). Thus, the best maneuver for object 1 was found at maneuver 21, object 2 at maneuver 41. With a hill climb factor of 20, this is the absolute fastest that a solution can be found.

To demonstrate the utility of learning maneuvers as opposed to routes, the scenario was changed to present similar SAM geometries, but at different locations. To a route planner, this would be a totally new scenario, but to MAVERICK, it is essentially the same as previous scenarios.

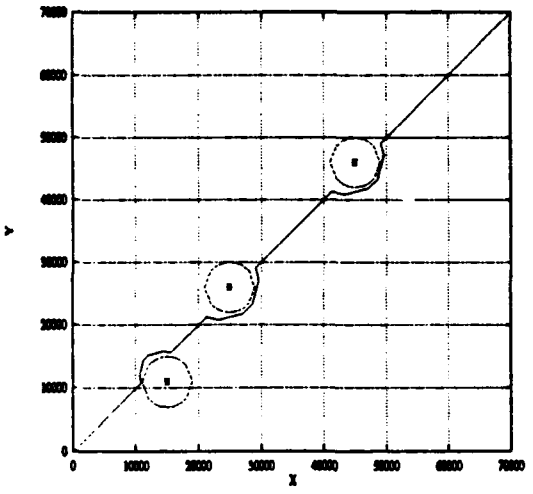


Figure 5.19. Best Maneuver, Long-Term Memory Active

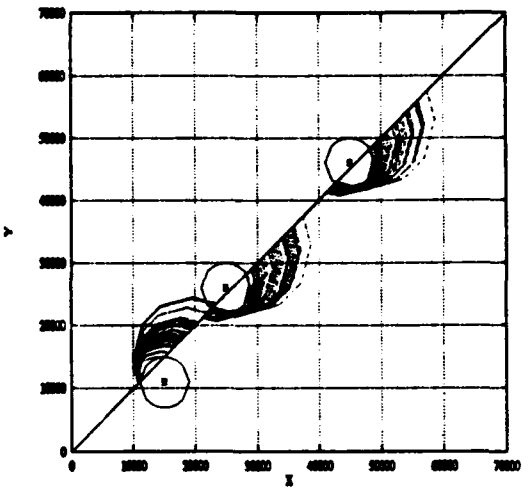


Figure 5.20. Route History, Long-Term Memory Active

Figures 5.19 and 5.20 show the maneuver solution and route history files for this modified scenario. An additional SAM with the same geometry was also added. Using long-term memory, MAVERICK was able to use the learned maneuvers to immediately find good maneuvers for each object. Even though the scenario appears different, the geometry is identical to the learned scenario, so MAVERICK can find very quick solutions.

To further demonstrate this performance, the three SAM scenario was modified to provide a vertical mirror-image of the previous scenario. The learned maneuvers are based upon a certain direction of radar contact, but the maneuver can be easily mirrored to provide a maneuver applicable to different scenarios.

Figures 5.21 and 5.22 show the route history and maneuver solution files for this third scenario. The scenario looks even more different from the original scenarios, but MAVERICK is able to apply the learned maneuvers just as if the scenario was identical to the learned scenario.

The importance of these tests is that by using the generalized learned maneuvers, MAVERICK was able to find *immediate* solutions to a variety of scenarios that were different from the original

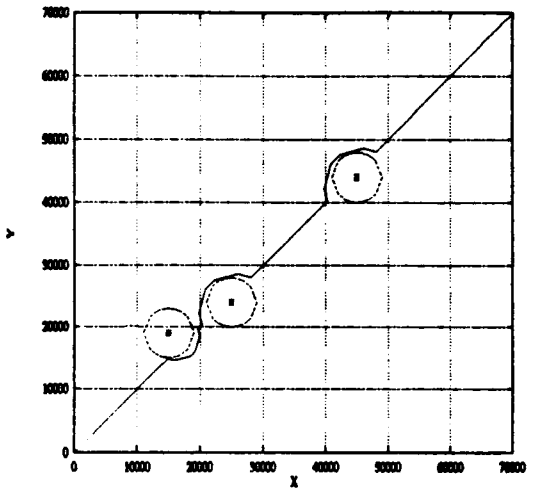


Figure 5.21. Best Maneuver, Long-Term Memory Active

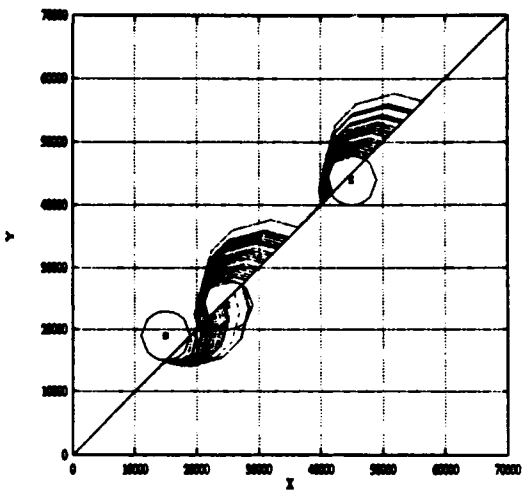


Figure 5.22. Route History, Long-Term Memory Active

learned scenario. The only time spent in exploration was for the hill climb factor, just to make sure the solution could not be improved.

5.6 Scenario Memory

The use of scenario memory is itself a heuristic. It assumes maneuvers learned from scenario objects will have utility when applied to unexplored objects. It also assumes maneuvers generated later in the exploration are more refined, and thus more useful for future exploration. Because this is a heuristic, and not a rule, it is sometimes wrong. Section 5.10.1 below shows an example of inappropriate application of scenario memory. Section 5.4 also discusses the negative effects of a particular application of scenario memory. The base case test used scenario memory; these results were presented in Figures 5.3 and B.2. The route history for the base case shows the difference in exploration between object 1 and object 2.

Figure 5.24 shows the similarity between object 1 and object 2's exploration. This is in direct contrast to figure 5.3. Also note the poor quality of the best maneuver generated in figure 5.23. In this particular case, scenario memory determined the difference between a good quality maneuver

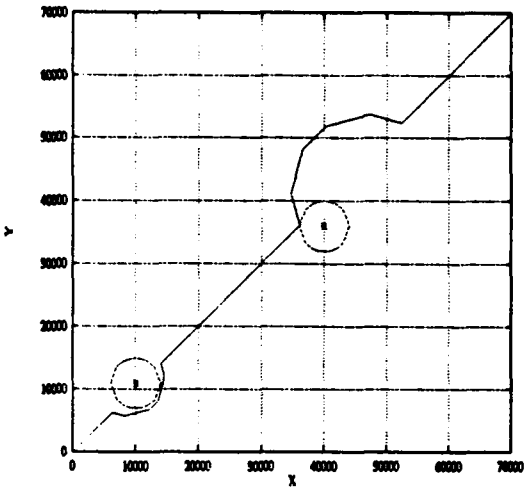


Figure 5.23. Best Maneuver, No Scenario Memory

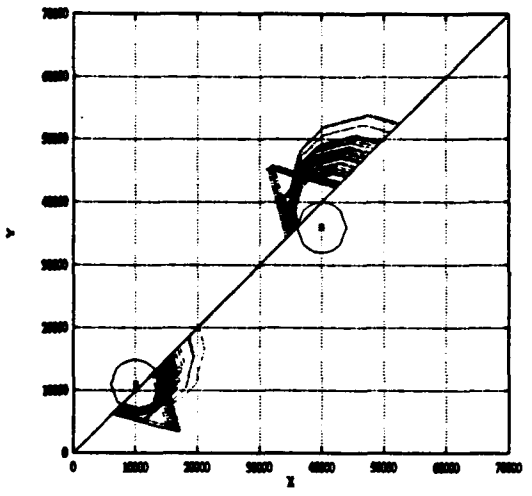


Figure 5.24. Route History, No Scenario Memory

and a poor one. In section 5.5, the negative effect of scenario memory was not to maneuver quality, it was to exploration efficiency. In section 5.4, some particular levels of hill climbing resulted in negative effects from scenario memory, while others had positive benefits from it.

The overall results of the tests involving scenario memory were generally inconclusive. The utility of the scenario memory varied from very high (the base case) to mixed (section 5.10.1) to negative (section 5.4). This suggests the scenario memory itself is useful, but the the heuristic which controls its application is weak. This is not surprising, considering the scenario memory is based upon the heuristic that all objects can benefit from the maneuvers learned from previous objects. A more sophisticated heuristic to control the application of scenario memory could possibly limit some of its negative effects while keeping the positive effects.

5.7 Heuristic Performance

One interesting behavior discovered during MAVERICK development was that the system seemed to be unusually tolerant of minor flaws in the heuristics. Presumably, the self-guided nature of the DBL process compensates for flaws in the maneuver heuristics, as long as the flaws

do not preclude the discovery of particular maneuvers. The first step in analyzing this effect was to eliminate the "if" conditionals from two of the most useful heuristics, the scale-up and scale-down heuristics. Later steps removed more conditions. As previously mentioned, the purpose of the conditions on these heuristics was to nudge the system away from wasteful exploration. The results under these tests were surprisingly positive.

During these tests, the number of maneuvers explored to find the best maneuver for each object was used as a benchmark of efficiency. For comparison, during the base case the system found the best maneuver for object 1 at maneuver 23, and the best maneuver for object 2 at maneuver 65. This means 23 maneuvers were explored for object 1 before the final best maneuver was found. After this, 20 more maneuvers were explored for the hill climb factor, so exploration of object 2 starts at maneuver number 44. 21 maneuvers were explored for object 2 before the best maneuver for it was found. After this, 20 more maneuvers were explored for the hill climb factor, so 85 total maneuvers were explored.

5.7.1 Two Conditions Removed During this test, two condition tests were removed, while scenario memory and heuristic sorting were active. Long-term memory was not active.

Figure 5.25 shows the best maneuver found under the test case, and Figure 5.26 shows the route history from this case. The route history clearly shows that the system spent more time exploring cases that involved positive radar contact, when compared to the base case in F igure 5.3. However, the ultimate solution found was virtually identical to the base case. In terms of efficiency, the results were mixed. In this test, the best maneuver for object 1 was found at maneuver number 30, compared with 23 for the base case. The best maneuver for object 2, however, was found at maneuver 58 for this test. Because 50 maneuvers were explored before MAVERICK ***started*** on object 2, this means the best maneuver was found after exploring only seven maneuvers. In terms of maneuvers explored, the performance of this test was degraded 30% for the first object, but improved by 67% for the second object.

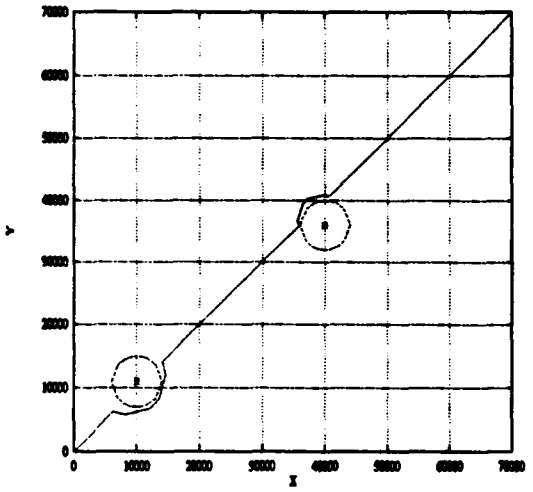


Figure 5.25. Best Maneuver, 2 Removed Conditions

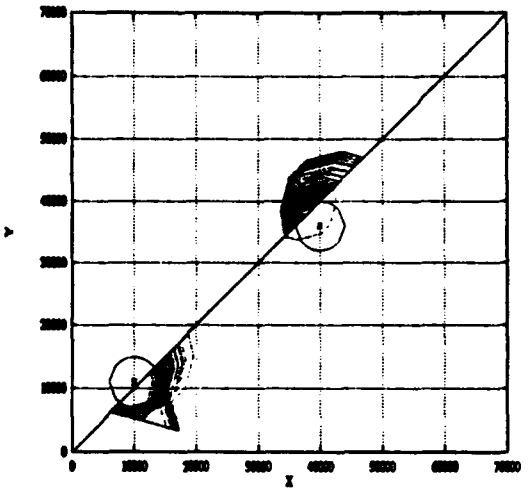


Figure 5.26. Route History, 2 Removed Conditions

5.7.2 Most Heuristic Conditions Removed In this test, all the conditions on all heuristics were removed, except for the mirror heuristic. Also, conditions that were necessary for program operation were retained.[3]

Figure 5.27 shows the maneuver solution found for this case, while figure 5.28 shows the route history. The results are almost identical to the previous case, with the exception of slightly less clutter around object 1 in Figure 5.28. However, the efficiency is slightly improved, with the best maneuver for object 1 found at number 15, and object 2 at number 58. In contrast to the previous test, this represents a 34% improvement for object 1's exploration and a 10% degradation for object 2's exploration, compared to the base case.

5.7.3 All Heuristic Conditions Removed In the final test, the mirror heuristic condition was removed. The reason for testing this case last is that the mirror heuristic is the most prone to generating maneuvers that appear good, but are extremely sub-optimal. These maneuvers can then be improved, in hill climbing fashion, taking up much of the MAVERICK's exploration time.

[3]For example, the "undelay" heuristic removes delays, so a delay must be present for this heuristic to make sense. The condition which checks this parameter could not be removed.

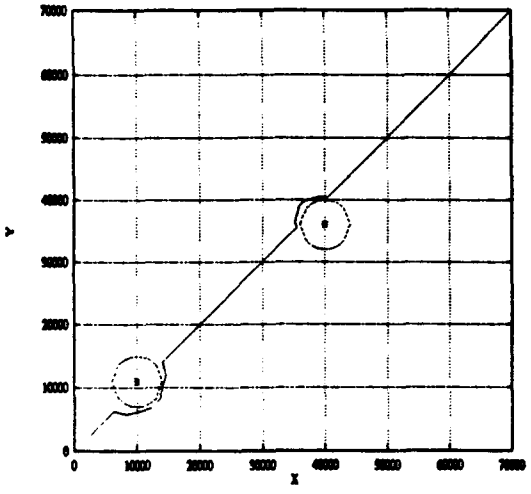


Figure 5.27. Best Maneuver, All But One Removed Conditions

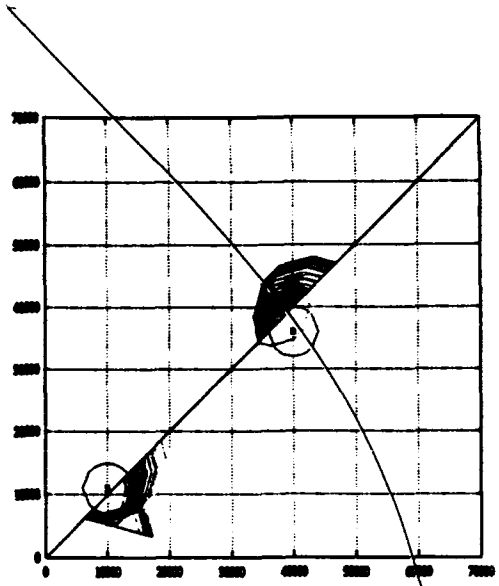


Figure 5.28. Route History, All But One Removed Conditions

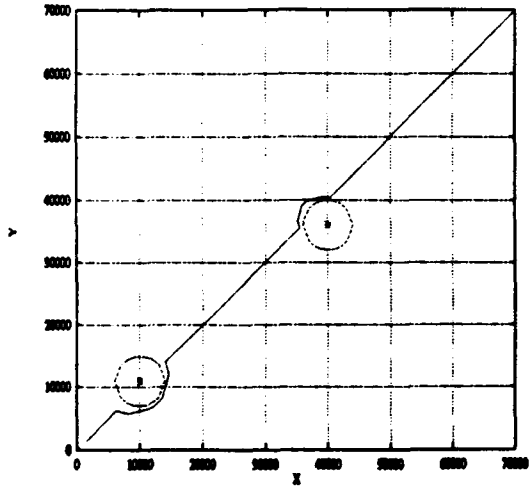


Figure 5.29. Best Maneuver, All Removed Conditions

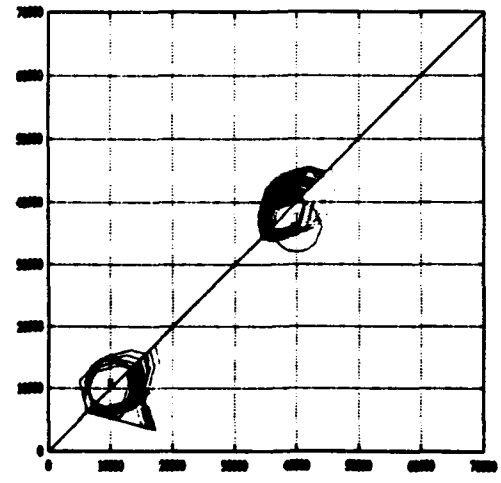


Figure 5.30. Route History, All Removed Conditions

Figure 5.29 shows the solution found is virtually identical to previous solutions.[4] The route history in Figure 5.29 shows the system did spend time exploring mirrored maneuvers that were not explored in the previous tests. The performance under this test was somewhat similar to the previous test; the best maneuver for object 1 was found at number 16, and object 2 at number 67. This represents a 30% improvement for object 1's exploration, and a 50% degradation for object 2's exploration.

The results presented were unexpected, and the evidence suggests the conditions on the heuristics were not uniformly useful, and in some cases were counterproductive. Removing the conditions on the heuristics improved the exploration performance in some cases, while degrading the performance in the other cases. The fact that there were *any* improvements in what should have been a degraded environment is unusual, however.

The conditions represent human biases (my biases) on what makes sense to explore. For example, I felt it generally made no sense to modify a maneuver with positive radar contact by having the aircraft turn tighter. The heuristic condition I imposed forced MAVERICK to only try turning tighter when radar contact was zero. My hypothesis for this behavior is that perhaps the conditions overly constrain MAVERICK to an expected search strategy; without the conditions MAVERICK finds unexpected paths to good solutions. This is currently only a hypothesis; it is difficult to trace the many maneuvers which leading to the final solution to find the "unexpected" deviation from the base case.

5.8 Faulty Maneuver Heuristic Performance

The previous tests used slightly degraded, but still correct heuristics, and resulted in good performance. The DBL process should be able to compensate for faulty heuristics, to a limited extent. The limitation arises from the nature of the heuristics. In some cases, a particular heuristic

[4] A close inspection reveals that none of the solutions are *completely* identical

may be necessary to lead to a good maneuver; in this case, if the heuristic is degraded or absent, the system performance will suffer. However, in other cases MAVERICK will be able to "work around" the faulty heuristics and arrive at a solution through a different path. This would be the case when superfluous heuristics were present.

The first test for defective heuristics involved a simple error that was encountered during MAVERICK development. In this test, the scale-up heuristic (H3) has a sign inverted, so it incorrectly acts like the scale-down heuristic (H2). In this situation, there is no scale-up heuristic, but the system compensates for this error and still finds a good solution. In these tests, scenario memory was active, long-term memory was inactive, heuristic sorting was active, and the hill climb factor was twenty.

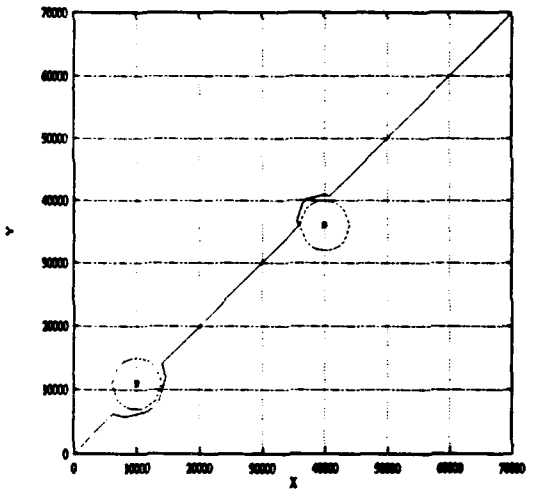


Figure 5.31. Best Maneuver, 1 Slightly Defective Heuristic

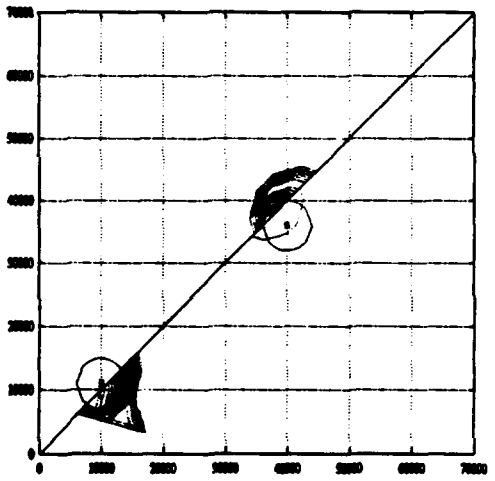


Figure 5.32. Route History, 1 Slightly Defective Heuristic

Figures 5.31 and 5.32 show the maneuver and route history files for this case. The maneuver solution is similar to the good maneuvers discovered in previous cases. The route history file highlights the defective operation of the heuristic. In the base case route history file (figure 5.3), the system spent almost no time exploring maneuvers that were within the radar coverage zone. In

this test, however, several maneuvers within the zone were tested because when MAVERICK tried to scale up maneuvers to get out of the radar zone, it ended up scaling them down instead.

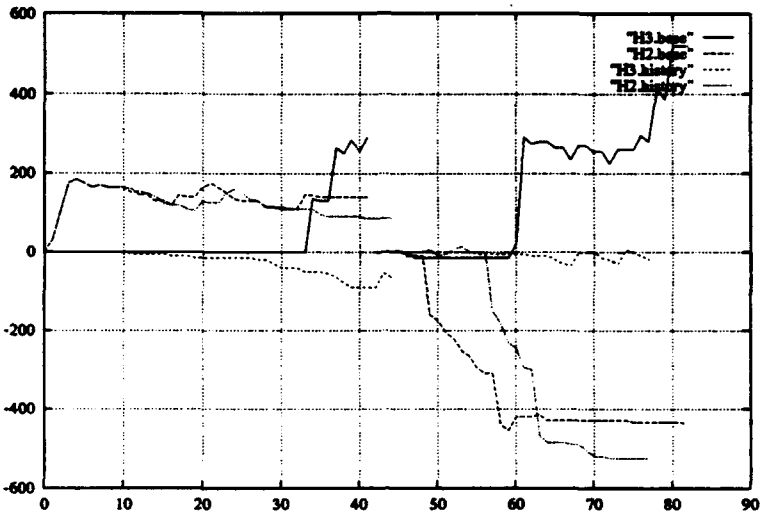


Figure 5.33. Heuristic History, Scenario Memory and No Scenario Memory

Figure 5.33 further highlights H3's defective behavior. This figure shows the heuristic history for the base case (H3.base, H2.base), and for this test. (H3.history, H2.history). H2's performance is similar in both the base case and this test, but H3's performance relative to the base case is severely degraded, demonstrating that it is not performing as it should. Presumably, H2's performance in the test is not identical to the base case performance of H2 because of the effects of the defective H3 heuristic.

To further test the system performance using degraded heuristics, the delay heuristic was modified to be non-operational. The system could still use the heuristic; it would simply have no effect.

Figures 5.34 and 5.35 show the maneuver and route history files for this test. The route history file differs sharply from the base case route history in Figure 5.3, highlighting the different exploration path used. The generated maneuver is certainly sub-optimal, but is fairly good considering the degraded heuristics. The significant fact is that MAVERICK degrades gracefully; small

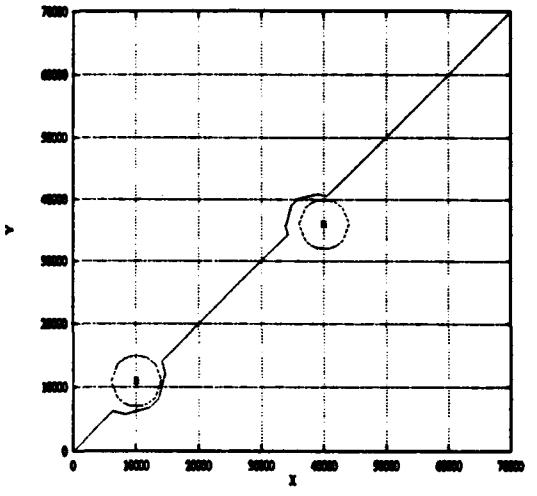


Figure 5.34. Best Maneuver, 2 Defective Heuristics

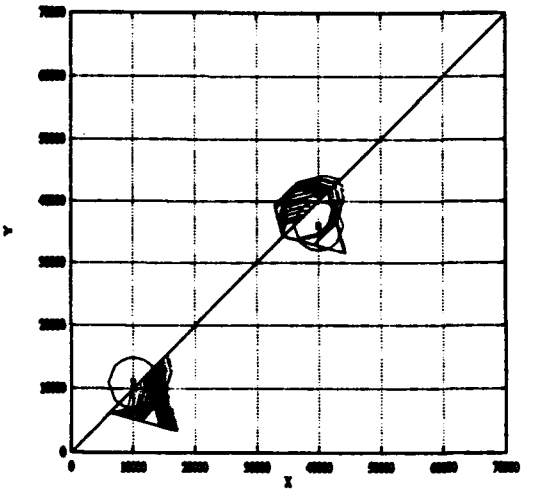


Figure 5.35. Route History, 2 Defective Heuristics

errors in the heuristics do not completely degrade the system performance. Because of the nature of DBL, the system finds the best solution given the tools (heuristics) it has available.

Figure 5.36 shows how the heuristic history could be used to track down defective heuristics. H9 is the heuristic that was effectively neutralized; note that in the heuristic history it remains zero for the entire exploration. The reason for this is MAVERICK never tests the same maneuver twice; since H9 did not change the maneuver, no maneuvers were ever tested using it. Also, the values of the other heuristics have sharply declined compared to the base case in figure 5.6. This is because many of the heuristics end up working "together": i.e., one heuristic is more useful after another heuristic has been applied. Because there were two defective heuristics in this test, the performance of the rest of the heuristics suffered.

5.9 Route Variability

All the previous tests involved a straight-line path as the initial route for the simulation aircraft. Because MAVERICK learns *maneuvers*, and not routes, the initial route for the simulation

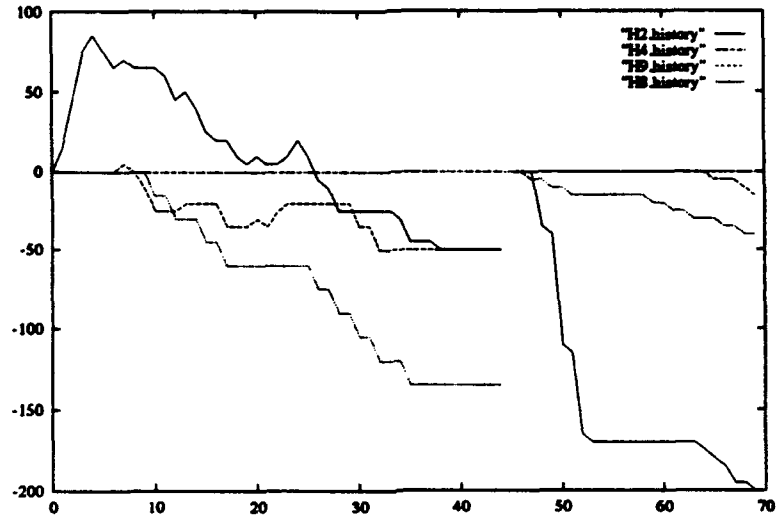


Figure 5.36. Heuristic History, 2 Degraded Heuristics

should be immaterial. To demonstrate this, the base case scenario was modified to use a base route involving several turns with the same placement of SAM's.

Figure 5.37 shows the best maneuver found for this scenario, a sub-optimal but reasonable maneuver. Figure 5.38 shows the route history for this test. The straight line path shown in this graph is a function of the operation of GNUPLOT; it is not an actual flown route. The performance of the MAVERICK under this scenario is similar to the performance under the base scenario, demonstrating MAVERICK can handle a variety of flight paths; a straight flight is not a requirement.

5.10 MAVERICK Limitations

The previous tests all involved the base case scenario. MAVERICK is able to handle a variety of scenarios, within certain limits. The following tests show the situations where MAVERICK's performance starts to degrade.

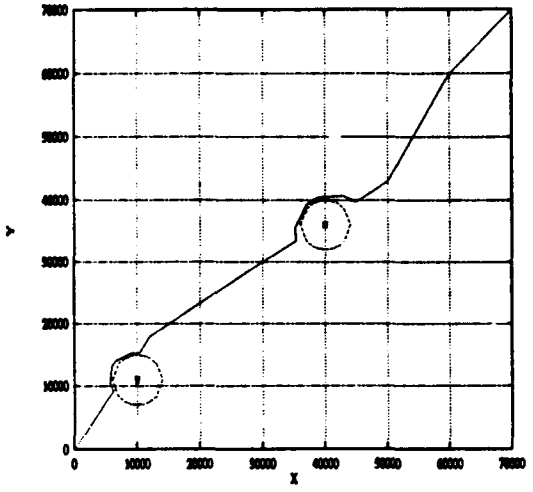


Figure 5.37. Best Maneuver, Non-Straight Route

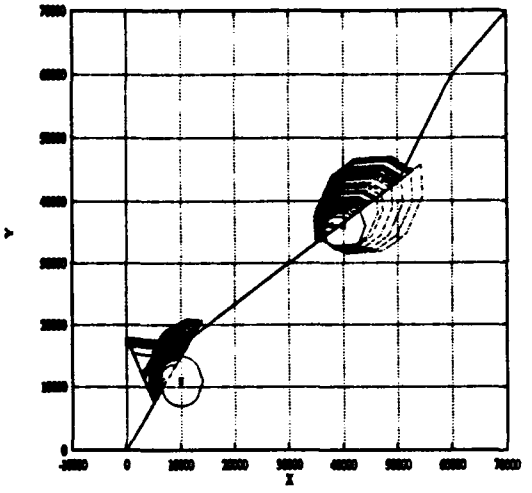


Figure 5.38. Route History, Non-Straight Route

5.10.1 Different SAM Site Ranges MAVERICK has no built-in constraints involving the ranges of the SAM sites in the scenario. However, because the aircraft in MAVERICK dynamically reacts to the threat objects during each maneuver test, the aircraft must be able to detect the SAM outside of the SAM's range. If the aircraft's range is less than the SAM's, it will never be able to find a zero radar contact solution. The SAM and aircraft sensor ranges are set up on the assumption that the aircraft can detect a weak radar signal before the SAM site could detect a returned weak signal, thus the effective range of the aircraft is typically larger than the SAM. In any case, if the aircraft doesn't detect the SAM until the aircraft is already within the SAM's range, then it is too late to do anything anyway.

To test MAVERICK's performance in a different scenario, the original scenario was modified to increase the range of the second SAM site. This test was performed with scenario memory, then without.

Figure 5.39 shows the maneuver solution found for this scenario, which is of good quality. The route history file in figure 5.40, however, shows the inefficiency of this exploration. MAVERICK attempted to use the maneuvers learned via the scenario memory, and took a long path to arrive

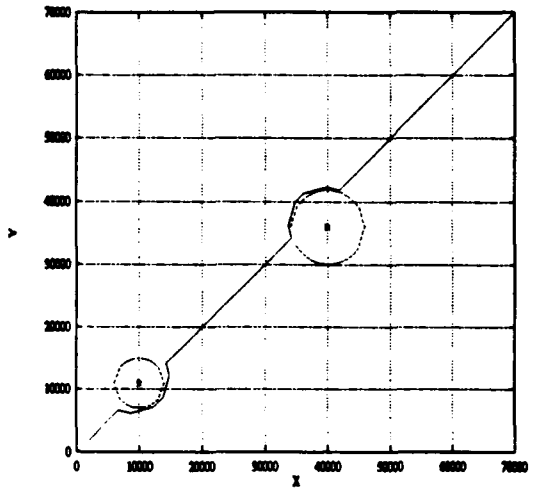


Figure 5.39. Best Maneuver, Scenario Memory ON, Scenario 2

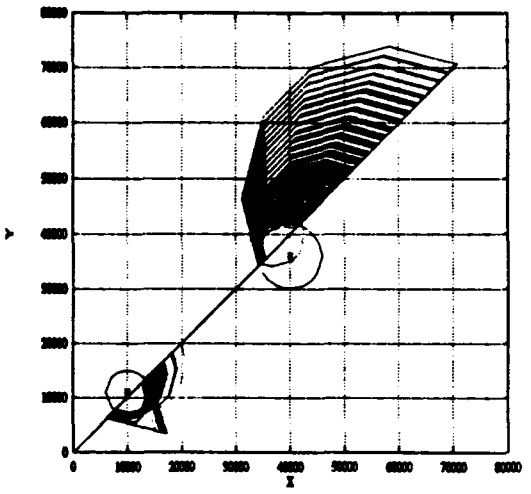


Figure 5.40. Route History, Scenario Memory ON, Scenario 2

at the solution. In this scenario, the learned maneuvers had negative utility. 160 maneuvers were explored before the maneuver for the second SAM site was discovered.

The second test involved deactivating scenario memory to determine the negative effects caused by it. Figure 5.41 shows the maneuver solution found was virtually identical. However, figure 5.42 is in direct contrast to the previous results. By starting from scratch on the second object, a solution was found much quicker, after only 68 maneuvers in this case. This demonstrates an inappropriate application of scenario memory, as the second object was different enough from the first object to cause the scenario memory to be counterproductive.

5.10.2 Non-Independent Objects One of the scoping assumptions made for MAVERICK was that objects could be reacted to independently. Put another way, the maneuver learned for one object does not affect the radar coverage for other objects. The reason for this assumption is the desire for generality of learned maneuvers. If two objects are close enough to cause interactions between their respective maneuvers, then MAVERICK ends up learning maneuvers for "sets" of objects, or for entire scenarios, thus limiting the generality and future utility of the learned maneu-

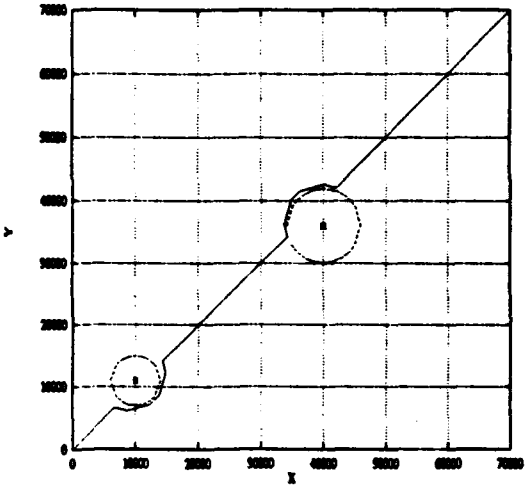


Figure 5.41. Best Maneuver, Scenario Memory OFF, Scenario 2

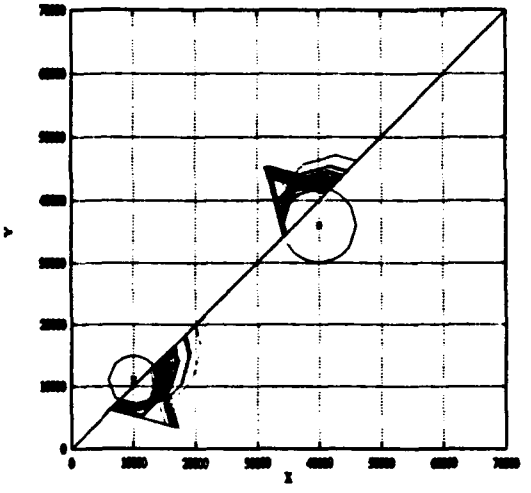


Figure 5.42. Route History, Scenario Memory OFF, Scenario 2

vers. This is not to say that such a capability would not be useful, it was just beyond the scope of this thesis.

Along with this assumption was another that there was some reason the aircraft should return to its original (straight-line) flight path. This was partially to allow for maneuver independence, but also to provide minimal trip-times. Because of this assumption, the maneuver heuristics are all defined in terms of returning to the original flight path vector. This causes problems when a second object is encountered while the maneuver for a previous objects is still being executed; the system uses the current (maneuver) flight vector as the vector to return to, providing erroneous maneuver execution. Because of the modular nature of MAVERICK, the capability to respond to non-independent objects could be added by creating a new set of maneuver heuristics; nothing else in the system would need to be changed.

An illustration of this potential problem is seen in Figure 5.20, as a result of the three SAM test. Close inspection reveals the maneuvers for the first SAM site do intersect the second SAM site slightly. Figure 5.22 shows an even more pronounced example of this potential problem. In these particular cases, the independence problem does not occur because of the sequence of MAVERICK

exploration. While MAVERICK is exploring maneuvers for object 1, the maneuvers for object 2 are always null. Thus, if the object ID's of the objects in the simulation are encountered in sequence,[5] the independence problem will not occur. This is not a real solution to the problem, it is just a method of limiting its effects.

5.10.3 Inadvertent Hill Climbing Figure 5.42 also shows another problem with MAVERICK's operation. It was mentioned in Chapter 2 that a DBL system is completely limited by its simulated world. Figure 5.40 is a prime example of a problem resulting from this fact. In some cases a maneuver that is scaled will result in slightly less radar contact when it should result in the same radar contact. The improvement in radar contact is only in fractions of a second, but it is still an improvement, so MAVERICK takes advantage of it. Also, the weighting of the interestingness heuristics is set up to make improvements in radar coverage more important that those in trip time. These factors combine together to produce the results in figure 5.40. In this case MAVERICK scales up a maneuver which results in slightly better radar coverage, so it keeps applying the same heuristic (because of heuristic sorting), and keeps getting slightly better results. Eventually this stops working, so MAVERICK starts scaling down some of those maneuvers to improve their trip time. This is shown by the clustering of similar maneuvers in the figure. Eventually MAVERICK converges upon a good solution.

The root cause of this inefficient behavior is an error in the simulation; maneuvers which should not be improving radar coverage are treated as if they are improving coverage. This highlights the limitations the simulator imposes upon the DBL process; MAVERICK can only be as good as the simulator it uses. The cause of this error was never determined; the errors are slight and are believed to be small variations in the route-points generated by RIZSIM. They could also be caused by inadvertent rounding in RIZSIM or in MAVERICK's radar calculation functions.

[5]MAVERICK does not "care" about the sequence of the object id's, they are just a convenient label. There is no built-in requirement for any sequence

This problem could be bypassed by limiting radar coverage improvements to a second or more, but this would compromise the granularity of the final solution to one-second increments.

5.11 Qualitative Issues

Some aspects of DBL were learned during development of MAVERICK, and these issues are difficult to quantify. DBL systems can be particularly difficult to develop compared to a conventional program. These difficulties are directly tied to the operation of the DBL process. For example, consider the case of convention computer program development. In this case, the programmer finds the causes of any unexpected behavior, then fixes the problem to cause the program to match his expectations of it. When a DBL system deviates from expected behavior, the programmer must first decide whether the deviation is truly a problem with the program, or whether it is a positive, but unexpected behavior of the program. If it is judged to be a problem, then it also frequently difficult to find the cause of the problem (this is also true in conventional programming, of course). The reason it is often difficult is because a DBL system does not operate in the sequential mode of a normal program. It jumps around in its exploration, and the "motives" of the system at any one time must be ascertained before deciding that the location of the problem has been found. Many of the history files presented were used to try to guess what the system was trying to do at any one time. To further complicate matters, corrections must be accomplished with extreme care to prevent compromising the independence of the DBL process. Although some correction will indeed fix a certain problem, they will, in effect, be "telling" the system what to do instead of telling the system how to find its own solutions. A normal programmer will use all the tools at his disposal to correct problems in a conventional program. A DBL programmer must operate without using all the tools at his disposal; he must find ways to correct problems without affecting independence.

5.12 Summary

5.12.1 Heuristic Sorting The results of sorting the heuristics versus not sorting them was about even over a large number of trials. However, the beneficial effect of removing the human bias from ordering them suggests that heuristic sorting is beneficial overall to the system

5.12.2 Scenario Memory Scenario memory had the most mixed results of any of the learning techniques. In some case it was necessary, in others it was a liability. A more sophisticated heuristic for determining the appropriate application of scenario memory is needed.

5.12.3 Long-Term Memory Long-term memory is very useful in scenarios where it is applicable. It can be applied to a variety of scenarios that are different in appearance, but similar in geometry, to the original learned scenario

5.12.4 Hill Climb Factor The hill climb factor generally gives better results at deeper levels of exploration, with a few exceptions. The exceptions are most likely caused by the scenario memory and not deeper exploration levels.

5.12.5 Heuristic Performance MAVERICK operates with a very simple set of heuristics, and the operation of these heuristics is fairly robust. The conditions on the heuristics appear to not be as significant as originally thought; without them MAVERICK sometimes performs better, and sometimes performs worse. MAVERICK also performs well with slightly faulty heuristics.

5.12.6 Overview MAVERICK's performance was fairly robust over a variety of different scenarios. Although the domain is extremely limited, the system does demonstrate some of the flexibility of the DBL process by adjusting to imperfect programming and different scenarios. Learning often provides a significant speed-up to the discovery process, but if applied inappropriately can have negative results. The decision of the appropriate time for use of the various learning techniques is a complex issue and is not easy to resolve with overly simple heuristics. Overall, the

system performed better than expected, considering its prototypical nature. MAVERICK demonstrated many of the positive aspects of DBL, as well as the negative aspects; the negative aspects may have more bearing on the eventual utility of DBL to a real system.

VI. Summary, Conclusions & Recommendations

6.1 Research Overview

This thesis presented MAVERICK, a Discovery-Based Learning (DBL) system designed to learn maneuvers in the route-planning domain. DBL was originally designed to learn in domains for which little domain knowledge exists. This thesis proposed using it in domains for which knowledge exists, but the utilization of this knowledge is difficult or time-consuming because of the knowledge acquisition bottleneck.

The DBL process is somewhat similar to the scientific discovery process: the system performs an experiment on a simulated world, gathers data about the experiment, then repeats the process with a new experiment. A DBL system manipulates and learns about concepts; these concepts are tested on the simulated world to determine their interestingness. The DBL system uses heuristics to manipulate these concepts with the goal of finding the most interesting concepts.

In MAVERICK, concepts are represented as maneuvers. The simulated world is the general-purpose battlefield simulation, RIZSIM. The agents used in RIZSIM are one aircraft and multiple SAM sites. The "maneuver-concepts" are manipulated to find the most interesting maneuver. In MAVERICK, this leads to the most efficient maneuver which allows the aircraft to safely negotiate the threat objects in the scenario.

Various learning techniques were utilized in MAVERICK to allow the system to learn from heuristic application, from objects within a scenario, and from different scenarios. The goal of this learning was to improve MAVERICK's performance over time.

The DBL process (as implemented in MAVERICK) was characterized, with special attention to the effectiveness of the various learning techniques used during exploration. The effects of the learning techniques were mixed; most had some utility, but the appropriateness of their application

to a particular case is still an open issue. The system did exhibit some robustness in terms of working around defective heuristics and heuristic conditions.

6.2 Conclusions

The overall goal of this thesis was to investigate the utility of applying DBL to a representative problem in the domain of pilot tactics. I believe MAVERICK demonstrates many of the positive and negative aspects of this issue, so this goal has been accomplished. To provide details of my conclusions, the objectives presented in Chapter 1 will be addressed individually.

- Determine if the domain knowledge "built-in" to a simulator is sufficient to guide a discovery process.

At first glance, the answer to this question would be yes. The heuristics were quite simple; MAVERICK could not discover good maneuvers with them alone. Some knowledge present in RIZSIM was exploited to guide the discovery process. The difficulty arises in determining the degree of knowledge in the simulator which is "used" by the DBL system. MAVERICK seems to exhibit the behavior of knowing certain pieces of knowledge that were not present in the system itself, such as:

1. The shortest distance between two points is a straight line.
2. An arc is the shortest path around a SAM site.
3. The minimum deviation from the planned course provides the most efficient path.

These items were not explicitly coded anywhere in MAVERICK, although the maneuvers generated seem to show MAVERICK uses this knowledge. Presumably, this knowledge could only have come from RIZSIM's model of the physical world. However, just because MAVERICK exhibits a behavior does not mean it "knows" a piece of knowledge. At best, even if it has learned this

knowledge, it learns it at a very shallow level. There is no underlying knowledge supporting why the above items are true. The other possibility is that the heuristics *implicitly* contain this knowledge, generating the explicit behavior when they are used with RIZSIM. I feel this possibility is doubtful because the heuristics are so simple; they are just "things to try." MAVERICK puts them together in unexpected and unplanned ways to generate good maneuvers.

The issue of defining what constitutes "new knowledge" to a system has always been a difficult problem for machine learning researchers. This may be a philosophical issue; if the system exhibits behavior which is useful, then it can be put to practical value. MAVERICK exhibits the behavior of having knowledge that is not explicitly coded into the system, so in the big picture, I feel the answer to this objective is yes.

- Determine if the maneuvers generated are of sufficient generality to apply to a variety of scenarios.

With the evidence presented in Chapter 5, the answer to this question would definitely be yes. As discussed, the *appropriate* application of general maneuvers is still an open issue, but the maneuvers generated have been demonstrated to apply to a variety of different scenarios, and different objects within the same scenario. This is in contrast to typical route-planner, which plans routes for a *specific* scenario.

- Determine how the limiting effect of the simulator fidelity affects the generated maneuvers.
- Determine if a discovery process can generate maneuvers that are applicable to real life.

These two objectives are closely related. The limiting nature of the simulator has been discussed. There are some real-world aspects of route planning that MAVERICK never considers:

1. The effect of various types of turns on fuel usage
2. The different radar technologies, and how they affect radar coverage

3. Other types of sensors (men on the ground, for example)

4. The third dimension (altitude) and how it affects radar coverage

These are just a few of the factors which are not considered for MAVERICK maneuvers, because they are not part of its simulated world. For example, SAM's in RIZSIM are essentially 2-dimensional, so a maneuver learned using RIZSIM would probably not work as well in a 3-dimensional world. To answer the first objective, the limiting effect of the simulator is severe; a DBL system is only as good (although it could be worse) than its simulator.

This leads to the second question. The maneuvers generated by MAVERICK are usually quite applicable to RIZSIM, in many cases they work quite well. While RIZSIM is a fairly sophisticated simulator, compared to the real world it is low fidelity. Thus, the maneuvers generated by MAVERICK have limited utility in the real world. Given a more sophisticated simulator, the system should generate more applicable maneuvers. If the simulator is fairly close to the real world, then the generated maneuvers will be closer to a real world maneuver. It is worth noting that simulation fidelity is also a potential limitation to the training of human pilots.

- Characterize the performance of the DBL process

The intent of this objective was to investigate the performance of the DBL process in terms of:

1. Learning general maneuvers

2. Applying these generalized maneuvers to improve performance

Chapter 5 presents qualitative and quantitative data regarding these two items. The performance was better than expected for a prototype system but brought out many of the negative issues of DBL that affect its overall utility to real world problems

The maneuver generality worked well with the long-term memory to provide performance gains over a variety of scenarios. The generality was less useful when used with scenario memory to provide performance boosts within a scenario. The hill-climb factor did prevent the system from using quick, sub-optimal solutions but this was at the cost of an extra amount of exploration after a best solution had been found. Heuristic sorting provided gains in specific solutions, but overall was not particularly effective. Its strongest benefit is the removal of human biases from the heuristic order.

6.3 Overall Impressions

DBL has many strong points that make it a useful choice for specific machine learning applications. Its ability to learn with limited knowledge, graceful degradation, and independent operation give it some significant advantages over other methods. However, these advantages are overshadowed by the potential hurdles to a DBL system, especially one that will be critical part of a Pilot's Associate type application. I see the potential problems with using DBL as follows:

Simulator Availability MAVERICK highlights the limiting nature of the DBL simulator. RIZSIM was made to be modified, most simulations are not designed for this purpose. A DBL system must be able to "plug in" to the simulator effectively, or it will not be able to operate to its full potential. Added to this is the lack of availability of appropriate simulators for different aspects of the learning domain. Having to write a simulator just for a DBL system would quickly nullify many of the gains realized from its small requirement for domain knowledge.

Development Difficulty As discussed in Chapter 5, the development of a DBL system requires a different paradigm than conventional computer program developments. The question of "How much guidance is too much guidance?" will always be a significant issue for the development of a DBL system.

Validation & Verification Because DBL is an inductive process, its conclusions will typically have no formal reasoning behind them. In other words, it will not be able to explain why it reached a solution, the user will just have to accept it. However, this is also true of other AI systems such as neural nets, genetic algorithms, and many pattern recognition systems. While this can be useful for some domains in which any answer is better than no answer, in critical systems it is usually not acceptable. This would relegate the usefulness of such a system to a position where it was simply giving advice, which could be accepted or rejected by a human judge. Without a means of verification, it could never be trusted to act autonomously.

Speed One problem that will always plague a DBL system is speed. It uses a simulator to develop interesting, and ideally more useful solutions than a "hard-wired" system, but it does this at the sacrifice of speed. The interface between the learning system and the simulator, plus the simulator overhead will usually slow down the speed of the system compared to a system which operates on its own. The learning aspects of DBL do improve its performance, but this is always within the sphere of DBL performance.

In summary, I feel that while DBL represents a viable means of eliminating the knowledge application bottleneck in some cases, it has no application to the active (during the mission) role of the PA, for the reasons described above. It could have a limited role in areas where speed and verification are not a requirement, such as an on-the-ground pre-mission planner. In this role, it could learn tactics for scenarios which could be suggested as possibilities for pilots, with the pilot acting as the final judge. The learned tactics could also be used by an "enemy agent" in simulated combat, where verification of the appropriateness of the tactics is not an issue.

6.4 Future Research

There are a few areas where MAVERICK could be expanded to provide a more thorough understanding of the DBL process. These areas are as follows:

Improved Simulation RIZSIM could be improved to provide a higher fidelity model, or another high fidelity model could be used. This would help establish the relationship between maneuver quality and simulation quality.

Scenario Memory Improvements There is evidence that scenario memory could be more universally useful if a scheme was developed to determine a more appropriate method of application. A higher fidelity simulation may also improve the effectiveness of scenario memory because more "features" of the simulation could be learned for application to later scenario objects.

New Domains MAVERICK could be applied to a new domain by simply changing its heuristics and the interface functions. The underlying structure would remain the same. It could be applied to any domain for which a simulator exists. This could provide more data about the applicability of DBL to different domains.

6.5 Summary

Overall, the investigation of DBL through MAVERICK proved enlightening. Even though MAVERICK worked in a very limited domain, several positive aspects of DBL were demonstrated and explored. MAVERICK demonstrated a suprising robustness and was able to operate effectively in a wide variety of situations. Negative aspects of DBL were also encountered, both as a result of MAVERICK's operation and experience gained during its development. MAVERICK exhibited a pronounced tendency toward unpredictability, complicating its operation and its development process. This likely precludes DBL from application to critical systems. However, the positive aspects of DBL suggest it has a significant potential for utility to other systems.

Appendix A. *Radar Coverage Calculations*

RIZSIM does not provide the capability to calculate the total coverage (time) that an object spends in another object's sensor range. RIZSIM is supposed to schedule a route-point at the entry-point to a sensor range, but in practice this did not happen reliably. In any case, the time of entry and the time of exit were required to determine the total time the aircraft was in radar coverage of a SAM. To minimize modifications to RIZSIM, these calculations were performed within MAVERICK. It is important to note these calculations were completely removed from the learning component of MAVERICK, they just provided an added capability to RIZSIM.

Although RIZSIM does not provide this information directly, the components within the **sensor_check** module do this calculation as part of RIZSIM's normal operation. Rizza'a thesis (20) provides a clear explanation of the details of these calculations, and much of my code was generated from his equations (not from the C code). The calculations done in MAVERICK are a simplified version of the algorithm Rizza uses; for more detail on the full algorithm, see (20).

The basic operation of the algorithm takes a 2-dimensional velocity vector, a starting coordinate, a second object's coordinate, and a second object's sensor range, and calculates the time at which the first object enters the second object's sensor range.

In my implementation, the following components are used:

1. x_{at1}: the x coordinate of the aircraft at time t_1
2. y_{at1}: the y coordinate of the aircraft at time t_1
3. v_{axt1}: the x velocity vector of the aircraft at time t_1
4. v_{ayt1}: The y velocity vector of the aircraft at time t_1
5. x_{st1}: the x coordinate of a SAM site
6. y_{st1}: the y coordinate of a SAM site

Note the velocities of the SAM sites are zero; this simplifies many of Rizza's calculations, which were designed for two moving objects.

$$d = \sqrt{(x_{at} - x_{st})^2 + (y_{at} - y_{st})^2} \tag{A.1}$$

Equation A.1 show the standard distance equation applied to an aircraft and a SAM at any time t. The x coordinate of the aircraft after a given Δt is given by $x_{at} = x_{at1} + v_{xa}\Delta t$, and similarly for the y coordinate. Then

$$x_{at} - x_{st} = (x_{at1} + v_{xa}\Delta t) - x_{st1} \tag{A.2}$$

And

$$y_{at} - y_{st} = (y_{at1} + v_{ya}\Delta t) - y_{st1} \tag{A.3}$$

Substituting equation A.1 into the above equation yields:

$$((x_{at1} - x_{st1}) + v_{xa}\Delta t)^2 + ((y_{at1} - y_{st1}) + v_{ya}\Delta t)^2 = d^2 \tag{A.4}$$

Letting $\Delta x = x_{at1} - x_{st1}$, $\Delta y = y_{at1} - y_{st1}$, and putting into the form of a quadratic equation yields:

$$(v_{xa}^2\Delta t^2 + (2\Delta x v_{xa} + 2\Delta y v_{ya})\Delta t + ((\Delta x)^2 + (\Delta y)^2) - d^2 = 0 \tag{A.5}$$

Thus,

$$\Delta t = \frac{-(2\Delta x v_{xa} + 2\Delta y v_{ya}) \pm \sqrt{(2\Delta x v_{xa} + 2\Delta y v_{ya})^2 - 4(v_{xa}^2 + v_{xa}^2)((\Delta x)^2 + (\Delta y)^2 - d^2)}}{2(v_{xa}^2 + v_{ya}^2)} \tag{A.6}$$

Using equation A.6, and substituting the SAM's sensor range for d, the time of sensor contact can be calculated. Non-imaginary solutions are contact points, showing the enter and exit times

for the SAM. Imaginary solutions indicate there will be no sensor contact, or the object is already in sensor contact.

This equation is embodied in the **calc** function, which inputs a Δx, a Δy, a v_{xa}, a v_{ya}, and a sensor range (d), and returns a list consisting of the two roots found, or NIL if non-imaginary roots are not found. The location of the SAM's is already known, by the initial read from the data file.

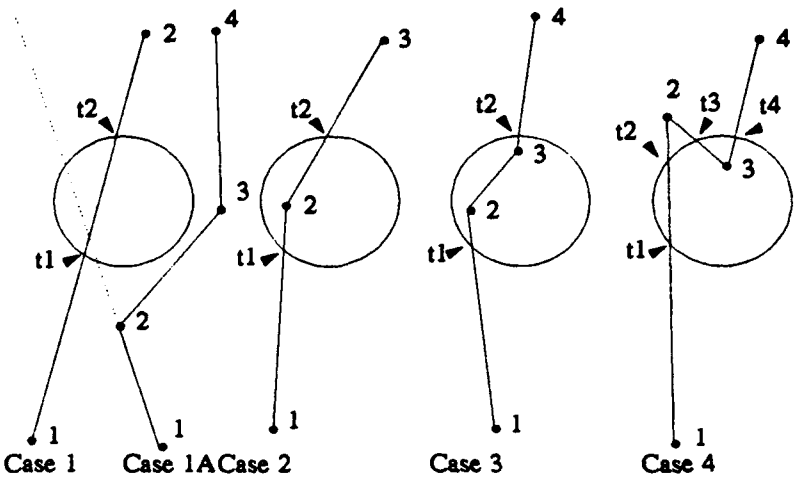


Figure A.1. Radar Coverage Cases

Figure A.1 shows the variety of cases the radar calculation functions must consider. The simplest case is Case 1, where the vector simply crosses the radar zone in a straight line. Using the 1→2 vector, the **calc** function will return t_1 and t_2. The first complication arises in Case 1A. The problem is that the 1→2 vector will yield a valid (non-imaginary) solution, because it assumes the continued vector given by the dotted line. In this case, the route list is "cdr'd" down to discover there is no radar coverage for this maneuver. Case 2 is also more complex. In this case, the 1→2 vector must be used to determine t_1, while the 2→3 vector must be used to determine t_2. As before the 1→2 vector will give a valid exit time, although the 2→3 vector will not give a valid enter time, because it is already in coverage. The next case is even more complex; here the 1→2 vector is used to determine the enter time, the 2→3 vector is skipped, and the 3→4 vector determines the exit

time. Finally, case four provides the maximum complexity, although it doesn't occur often. In this case, the 1→2 vector gives enter and exit times, which provide a delta to add to the 2→3 and 3→4 generated delta. The **enter** and **exit** functions in calc.lisp source code file are used to handle the various cases encountered.

Appendix B. *MAVERICK Source Code*

This appendix includes the LISP source code for MAVERICK

B.1 Maverick Main Functions

These functions comprise the central DBL process and MAVERICK functions.

B.1.1 Maneuver Heuristics These heuristics suggest new maneuvers.

B.1.2 Interestingness Heuristics These heuristics determine the interestingness of a maneuver.

B.1.3 Maneuver Transforms These functions transform a maneuver or move from one form to another.

B.2 Class and Method Definitions

Functions for calculating SAM radar coverage are on pages B-42 – B-48.

Utility functions are on pages B-21 – B-24 and B-26 – B-34.

```
;;;; MAV.LISP  Main program for the MAVERICK Discovery-Based Learning
;;;; System.  See Documentation for details on program operation
;;;; Definitions: Move:  a maneuver for a certain object.  A maneuver
;;;; is composed of a number of moves, one for each object
;;;;              Best maneuver:  The maneuver with zero radar contact
;;;;              that has the shortest trip time.  If there is no
;;;;              maneuver with zero radar contact, there is no best
;;;;              maneuver.  There can also be a best move for a given
;;;;              object with the same qualifications.

;;; Load the support files needed for MAVERICK
(load "class.lisp")
(load "utils.lisp")
(load "calc.lisp")
(load "mav-utils.lisp")
;;;; System Global Variables

(defvar *active* nil "Determines if maneuvers are displayed as they are run")
(defvar *sort-heuristics* t "Use intelligent heuristic application")
(defvar *radar-weight* 50 "Relative interestingness weight of radar reductions")
(defvar *man-weight* 10 "Relative interestingness weight of man-time reductions")
(defvar *level-limit* 999 "Level (depth) limit for exploration")
(defvar *double-plus-good* 30 "Relative interestingess weight for both reductions")
(defvar *zero-radar* 1 "Relative interestingness weight for zero radar contact")
(defvar *best-bonus* 30 "Bonus for being the best manuver")
(defvar *maneuver-counter* 1 "Counter of the number of maneuvers explored")
(defvar *hill-climb-factor* 20 "Number of extra maneuvers explored for hill-climbing")
(defvar *memory* nil "Use remembered maneuvers or not?")
(defvar *smemory* t "Scenario memory or not?")
(defvar *best-heuristics* nil)
(defvar *history* t "Whether to write history data or not")
(defvar *master-move-list* '((0 0)) "List of all maneuvers tested")
(setf *print-length* 20) ;default is 10 for CL



;;; Instantiate the agenda

(setf master (make-instance 'agenda))

;;; Instantiate the root concept

(setf adam (make-instance 'maneuver
                          :interestingness 1
                          :parent '()
                          :move '((1 0)) ))



;;;; Function TRANSFORM-MANEUVER
;;;; This function is the 'executive' function for all the maneuver
;;;; transforms.  It is used to call the various transforms and keep
```

```
;;;; accounting for which transforms and heuristics are used.  It has
;;;; the following inputs:
;;;; 1.  Maneuver:  This is the maneuver which will undergo
;;;; transformation
;;;; 2.  Number:  The maneuver for the nth object is to be transformed
;;;; 3.  Transform:  Which transform function to call
;;;; 4.  Heuristic:  Which heuristic to attribute to this transform
;;;; 5.  Optional: arguments required by various transforms
(defun transform-maneuver (maneuver number transform heuristic
                                    &optional t-argument1 t-argument2)
  (let ((new-move (copy-seq (move maneuver)))

        )
        (setq transformed-move
              (cond ((null t-argument1)
                     (funcall transform (my-nth number (move maneuver))))
                    ((null t-argument2)
                     (funcall transform (my-nth number (move maneuver)) t-argument1))
                    (t
                     (funcall transform (my-nth number (move maneuver)) t-argument1 t-argument2))

      (setf (nth (- number 1) new-move) transformed-move)
      ;; Test to see if we have already tried this maneuver
      ;; if they are equal, then the move is already there
      (if (equal *master-move-list*
                 (adjoin new-move *master-move-list* :test #'equal))
          '()
        (progn
          (insert-maneuver new-move maneuver heuristic)
          (decf (interestingness maneuver) .001)  ;to force sorting
          ))))



;;;; Transform T-SCALE
;;;; This transform simply scales a maneuver positive or negative by
;;;; scaling all the durations within it.
(defun t-scale (move &optional (magnitude 1))
  "Scaling transform.  Adds to every duration"
  (cond ((null move)
         '())
        ((zerop (cadr move))
         move)
        ((AND (zerop (car move))
              (equal 1 (cadr move)))
         move)                          ;null move, ignore

        ((zerop (car move))
         (append (list (car move) (cadr move))
                 (t-scale (cddr move) magnitude)))
        (t
         (append (list (car move) (max 1 (+ magnitude (cadr move))))
```

```
                (t-scale (cddr move) magnitude))) ))


;;;; Transform T-MIRROR
;;;; This transform mirrors a maneuver by inverting the signs of all
;;;; the turns in a maneuver
(defun t-mirror (move)
  "mirror image of a move"
  (cond ((null move)
         '())
        ((AND (zerop (car move)) (equal 1 (cadr move)))
         move)
        ((zerop (cadr move))
         move)
        (t
         (append (list (- (car move)) (cadr move))
                 (t-mirror (cddr move)) ))))


;;;; Transform T-SET-ANGLE
;;;; This transform adds a given angle and duration to the beginning
;;;; of a maneuver
(defun t-set-angle (move &optional (angle 0) (duration 1))
"used to set a determined angle and duration consed to a maneuver"
  (append (list angle duration) (cddr move)))



;;;; Transform T-TURN-RETURN
;;;; This transform initiates a turn of a given angle for a given
;;;; duration, the returns to the original path
(defun t-turn-return (move &optional (angle 0) (duration 1))
  "used to give a turn that is followed by a return to path"
  (let* ((reciprocal (- (* 2 angle)))
         )
  (append (list angle duration reciprocal duration) (cddr move))))



;;;; Transform T-SET-MOVE
;;;; This transform explicitly sets the maneuver
(defun t-set-move (move replacement-move)
  "copy a move number to other place"
  replacement-move)



;;;; Transform T-CLIP
;;;; This transform 'clips' a maneuver by taking a shortcut across
;;;; legs of the maneuver.  It is called by T-TRIM-MAX below
(defun t-clip (move &optional angle duration)
```

```
"clips a triangle"
(cond ((null move)
       '())
      ((AND (<= 4 (length move))
            (< 5 (second move))       ;minimum duration for trim
            (< 15 (abs (third move))));minimum angle to trim

       (let* (
              (angle1 (first move))
              (angle2 (third move))
              (point-angle (- 180 (abs angle2)))
              (angle (/ (- 180 point-angle) 2))
              (new-turn-angle (cond ((and (plusp angle1)
                                          (plusp angle2))
                                     angle)
                                    ((and (minusp angle1)
                                          (minusp angle2))
                                     (- angle))
                                    ((plusp angle1)
                                     (- angle))
                                    (t
                                     angle)))

              (int-angle (abs new-turn-angle))

              (clipped-duration (if (null duration)
                                    (/ (cadr move) 2) duration))
              (duration1 (second move))
              (duration2 (fourth move))
              (cut-duration1 (- duration1 clipped-duration))
              (gamma (- 180 (+ point-angle int-angle)))
              (third-angle (if (plusp new-turn-angle) gamma (- gamma)))
              (factor (/ cut-duration1 (my-sin gamma)))
              (cut-duration3 (* factor (my-sin int-angle)))
              (duration3 (- duration2 cut-duration3))
              (new-duration2 (* factor (my-sin point-angle)))
              )

         (list (float angle1)
                       (float clipped-duration)
                       (float new-turn-angle)
                       (float new-duration2)
                       (float third-angle)
                       (float duration3) )))
      (t
       move) ))


;;;; Transform T-TRIM-MAX
;;;; This function is used to specialize the calling of T-TRIM, above.
;;;; Given a triangle maneuver, it clips it halfway across, parallel
```

```
;;;; to the line of flight, giving a trapezoid shape.  With a
;;;; trapezoid-shaped maneuver, it clips both edges.  After this, no
;;;; further clipping occurs
(defun t-trim-max (move)
  (let* ((angle-list (loop for item in move by #'cddr
                           do collect item))
         (pos-list (mapcar #'abs angle-list))
         (max-turn (unless (null move) (apply #'max pos-list)))
         )
   ;;to allow for delayed moves
    (if (zerop (car move))
        (progn
          (setq delay (list (car move)
                            (cadr move)))
          (setq move (cddr move)))

      (setq delay nil))

  (cond ((null move)
         '())
        ((> 4 (length move))
           move)
        ((AND (= 6 (length move))
              (equal max-turn (abs (third move))))
         (append delay
                 (append (t-clip (list (first move)
                                       (second move)
                                       (third move)
                                       (fourth move)))
                         (cddr   (t-clip (list (third move)
                                               (fourth move)
                                               (fifth move)
                                               (sixth move)))))))
        ((equal 4 (length move))
         (append delay (t-clip move)))
        (t
         (append delay move)))))


;;;; Transform T-DELAY
;;;; This transform delays the start of a maneuver by adding a zero
;;;; turn for n seconds to the beginning of it, or adding to an
;;;; existing delay
(defun t-delay (move &optional (delay 1))
  (cond ((plusp (abs (car move)))  ;no delay already here
         (append (list 0 delay) move))
        ((zerop (car move))
         (append (list 0 (+ (cadr move) delay)) (cddr move)))))
```

```
;;;; Heuristics.


;;;; Heuristic H1
;;;; This heuristic is use to establish the initial population of
;;;; maneuvers.  It starts a series of turn-and-return maneuvers
;;;; (triangles) starting from the appropriate side given the
;;;; direction of the radar contact
(defun h1 (maneuver &optional (force nil) (duration 40))
  "For now, flight 10% of the time. Force t to force application"
  (let* ((number (find-bad (radar-contact maneuver)))

         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))
        )
    (if force
        (progn

          (format t "running away <--------------%")
          ;; skip 0 angle turn
          (if (equal (third (my-nth needy (radar-directions maneuver)))
                     'l)
              (loop for i from -60 to -10 by 10
                    do (transform-maneuver maneuver needy 't-turn-return 'h1 i
                                           duration)))
          (if (equal (third (my-nth needy (radar-directions maneuver)))
                     'r)
              (loop for i from 60 downto 10 by 10
                  do (transform-maneuver maneuver needy 't-turn-return 'h1 i
                                         duration)))

          ))))


;;;; Heuristic H2
;;;; This heuristic scales down a move in a maneuver with zero radar
;;;; contact
(defun h2 (maneuver &optional (magnitude 1))
  "Scale it down"
  (let* ((number (find-bad (radar-contact maneuver)))
         (good-list (find-good (radar-contact maneuver)))
         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))

        )
        (if (zerop needy-radar)
            (progn
```

```
            (format t ">>>>>>>>>>> Scaling down ~D <<<<<<<<<<~%"
                    needy)
            (transform-maneuver maneuver needy
                                't-scale 'h2 (- magnitude))))))


;;;; Heuristic H2A
;;;; This heuristic attempts to speed up repetitive scaling by
;;;; applying a higher magnitude scaling
(defun h2a (maneuver)
  "determine scaling"
  (let ((h2-repeats (count-reps 'h2 (heuristics maneuver)))
        )
  (if (and (equal (car (heuristics maneuver)) 'h2)
           (< 2 h2-repeats)
           )
      (h2 maneuver h2-repeats))))


;;;; Heuristic H3
;;;; This heuristic scales up a move in a maneuver with positive radar
;;;; contact.
(defun h3 (maneuver &optional (magnitude 2))
  (let* ((number (find-bad (radar-contact maneuver)))
         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))
         )
    (if (AND (plusp needy-radar)
             (> 5 (count-reps 'h3 (heuristics maneuver))))
        (progn
          (format t "<<<<<<<<<<<<<<<< SCALING UPPPPP >>>>>>>>>>>>>>>>~%")
          (transform-maneuver maneuver needy 't-scale 'h3 magnitude)))))



;;;; Heuristic H4
;;;; This heuristic clips a move in a maneuver with zero radar contact
(defun h4 (maneuver &optional angle magnitude)
  "clip maneuver"
  (let* ((good-list (find-good (radar-contact maneuver)))
         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))
         )
    (if (zerop needy-radar)
        (progn
          (format t "Clipping ------> ~D~%" needy)
          (transform-maneuver maneuver
                              needy 't-trim-max 'h4 angle magnitude)))))
```

```
;;;; Heuristic H5
;;;; This heuristic was used to randomly pop the best maneuver to the
;;;; top of the agenda.  It is not particularly effective, and the
;;;; non-determinism was not compatible with tests and analysis.  It
;;;; is not used in the current implementation
(defun h5 (maneuver)
  "percolate best to top periodically"
  (if (AND (best))
      (setf (interestingness (best))
            (+ 20 (interestingness (best))))))



;;;; Heuristic H6
;;;; This heuristic determines when it woudl be wise to mirror a move
;;;; in a maneuver, based upon the raar contact direction
(defun h6 (maneuver)
  "When to mirror heuristic"
  (let* ((good-list (find-good (radar-contact maneuver)))
         (bad-number (find-bad (radar-contact maneuver)))
         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))
         (delta (- (total-radar-contact maneuver)
                   (total-radar-contact (parent maneuver))))
         )
    (if (or

          (and
           (equal (third (my-nth needy (radar-directions adam)))
                  'r)
           (minusp (find-first-turn (my-nth needy (move maneuver)))))

          (and
           (equal (third (my-nth needy (radar-directions adam)))
                  'l)
           (plusp (find-first-turn (my-nth needy (move maneuver))))))
        (progn

          (format t "<><><><><><>< Mirroring ~D <><><><><><><~%"
                  needy)
          (transform-maneuver maneuver needy
                              't-mirror 'h6)))))


;;;; Heuristic HX
;;;; This heuristic was used to collect the best maneuvers for each
;;;; object into one combined best maneuver.  It is not use in the
;;;; current implementaion
(defun hx (&optional (spawn nil))
  (let* ((max-objects (max-objects main-board))
         )
    (setq new-move
```

```
          (loop as i from 1 to max-objects
                collect
                (if (rbest i)
                    (my-nth i (move (rbest i))))))
    (nsubstitute '(1 0) nil new-move) ;take care of still bad
    (if spawn
        (progn
          (setq new-maneuver (spawn-child adam))
          (setf (heuristics new-maneuver) '(hx))
          (setf (interestingness new-maneuver)
                0)

          (setf (move new-maneuver) new-move)
          ))
    new-move))



;;;; Heuristic H8
;;;; This heuristic is used to delay the start time of a move in a
;;;; maneuver with zero radar contact
(defun h8 (maneuver)
  "delay or undelay the start time of a maneuver"
   (let* ((number (find-bad (radar-contact maneuver)))
          (good-list (find-good (radar-contact maneuver)))
          (needy (find-needy main-board))
          (needy-radar (my-nth needy (radar-contact maneuver)))
         )
     (if (AND t
           (zerop needy-radar)
           (not (null-manp (my-nth needy (move maneuver)))))
         (progn
           (format t ">>>>>>>>>>> Delaying ~D <<<<<<<<<<~%" needy)
           (transform-maneuver maneuver needy
                               't-delay 'h8 2)))))


;;;; Heuristic H9
;;;; This heuristic is use to remove delays that were initiated by
;;;; othe heruristics
(defun h9 (maneuver)
  "undelay"
  (let* ((number (find-bad (radar-contact maneuver)))
         (good-list (find-good (radar-contact maneuver)))
         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))
         )
    (if (AND
         (plusp needy-radar)
         (< 1 (cadr (my-nth needy (move maneuver))))
         (not (null-manp (my-nth needy (move maneuver))))
         (equal (car (my-nth needy (move maneuver))) 0))
```

```
        (progn
          (format t "???? Undelaying ~D ??????????~%" needy)
          (transform-maneuver maneuver needy
                              't-delay 'h9 -1))) ))



;;;; Interestingness Heuristic IH1
;;;; Gives bonus for having zero radar contact
(defun ih1 (maneuver)
  "did we reach zero contact on this maneuver?"
  (if (find-good (radar-contact maneuver))
      (incf (interestingness maneuver) *zero-radar*))))



;;;; Interestingness Heuristic IH2
;;;; Gives bonus for improving radar coverage, both total and for the
;;;; currently explored object.
(defun ih2 (maneuver)
  "are we improving our radar coverage?"
  (let* ((radar-list (trace-parent maneuver 'total-radar-contact))
         (delta (- (total-radar-contact (parent maneuver))
                   (total-radar-contact maneuver)))
         (needy (find-needy main-board))
         (needy-radar (my-nth needy (radar-contact maneuver)))
         (parent-needy-radar (my-nth needy (radar-contact
                                            (parent maneuver))))
         (ndelta (- parent-needy-radar needy-radar))
         (man-time (man-time maneuver))
         )
    (if (AND (plusp (my-round delta 1))
             (plusp (my-round ndelta 1)))
        (progn (incf (interestingness maneuver)
                     *radar-weight*)))))



;;;; Interestingness Heuristic IH3
;;;; Gives bonus for improvin trip time
(defun ih3 (maneuver)
  "are we improving our trip time?"
  (let ((time-list (trace-parent maneuver 'man-time))
        (delta (- (man-time (parent maneuver))
                  (man-time maneuver)))
        )

    (if (AND (not (equal (car time-list) (cadr time-list))) ;no improve
             (plusp delta) )
        (progn (incf (interestingness maneuver)
                     *man-weight*)))))
```

```
;;;; Interestingness Heuristic IH6
;;;; Gives bonus for passing the zero threshold
(defun ih6 (maneuver)
  "Passing 0 is significant"
  (if (not (equal maneuver adam))
      (if (> (length (find-good (radar-contact maneuver)))
             (length (find-good (radar-contact (parent maneuver)))))
                                        ;we have a new 0
          (progn
            (incf (interestingness maneuver) 100)))))



;;;; Interestingness Heuristic IH7
;;;; Gives extra bonus for improving both radar coverage and trip time
(defun ih7 (maneuver)
  "Both improvement is extra-special"
  (let ((radar-list (trace-parent maneuver 'total-radar-contact))
        (time-list (trace-parent maneuver 'man-time))
        (r-delta (my-round (- (total-radar-contact maneuver)
                              (total-radar-contact (parent maneuver))) 1))
        (m-delta (- (man-time maneuver)
                    (man-time (parent maneuver))) )
    (if (AND (equal radar-list (sort radar-list #'<))
             (equal time-list (sort time-list #'<)))
        (incf (interestingness maneuver) *double-plus-good*))))



;;;; Function UPDATE-HEUR-VALS
;;;; This function updates the values of the heuristics after each
;;;; test, base upon whether the interestingness improved or not
(defun update-heur-vals (maneuver)
 "update heur vals for what worked"
 (let* ((last-heuristic (car (heuristics maneuver)))
        (next-to-last-heuristic (cadr (heuristics maneuver)))
        (improvep (> (interestingness maneuver)
                     (interestingness (parent maneuver))))
        (delta (- (interestingness maneuver)
                  (interestingness (parent maneuver))))
        (scalep (or (AND (equal last-heuristic 'h2)
                         (equal next-to-last-heuristic 'h2))
                    (AND (equal last-heuristic 'h3)
                         (equal next-to-last-heuristic 'h3))))

        )
     (if (not (equal last-heuristic 'hx))
         (progn
           (incf (cdr (assoc last-heuristic heur-vals)) delta)
           (if (AND next-to-last-heuristic
                    (not (equal next-to-last-heuristic 'hx)))
               (incf (cdr (assoc next-to-last-heuristic heur-vals))
```

```
                    delta)) ) )
      ;; Writes the heuristic histories
      (if *history* (dolist (item *heuristics*)
                        (setq heur-string (string item))
                        (setq list-filename (append
                                      (coerce heur-string 'list)
                                      '(#\.)
                                      (coerce "history" 'list)))
                        (setq filename (coerce list-filename 'string))
       (with-open-file (my-stream filename
                                          :direction :output
                                          :if-does-not-exist :create
                                          :if-exists :append)
                                  (format my-stream "~D~%"
                                          (cdr (assoc item heur-vals))  ))))))

;;;; These functions were used to randomize heuristic order for tests

(setf *heuristics* '(h5 h6 h8 h9 h3 h2a h4 h1 h2))
(setf *heuristics* (mix-up *heuristics*))
(setf *heuristics* '(h1 h2 h2a h3 h4 h5 h6 h8 h9))



;;;; Function APPLY-HEURISTICS
;;;; This function applies the heuristics to a newly tested maneuver,
;;;; and adds new null moves for new objects that are found.  It also
;;;; checks to see if the level limit has been reached
(defun apply-heuristics (m)
  (let* ((heuristic-list (heuristics m))
         (num-contacts (length (radar-contact m)))
         (percent (+ 1 (random 100)))
         (sorted-heuristics (mapcar #'car (sort heur-vals #'> :key #'cdr)))
         )
  (if (> num-contacts (length (move m))) ;found a new object
        (progn

          (dotimes (x (- num-contacts (length (move m))))
                   (setf (move m) (append (move m) '((1 0)))))))
      (update-stats m main-board)
;     (ih1 m)
    (ih2 m)
    (ih3 m)
    (ih6 m)
    (ih7 m)
    (update-heur-vals m)

    ; Check the level limit breach
    (cond ((> *level-limit* (level m))
           (if *sort-heuristics*
               ;use dynamically sorted heuristics
               (dolist (item
```

```
                        sorted-heuristics)
                       (funcall item m))
              ;use the defined order of heuristics
              (dolist (item *heuristics*)
                      (funcall item m))))
            (t
             (setf (interestingness m) -10)))

    (check-done m main-board)
    (write-heuristic-history)

))



;;;; Initialize the heuristic values to zero
(defun init-heur-vals()
  (setf heur-vals (mapcar #'(lambda (x) (cons x 0)) *heuristics*)))



;;;; Shorthand for the base case radar
(defun base-case-radar()
  (radar-contact adam))



;;;; Function WRITE-LEARNED-MANEUVERS
;;;; This function writes the optimal maneuvers learned, plus the
;;;; straight-line radar coverage to a file for long-term memory
(defun write-learned-maneuvers (filename)
  (let ((best-maneuver (best))
        )
  (with-open-file (my-stream filename
                             :direction :output
                             :if-does-not-exist :create
                             :if-exists :append)

                  (dotimes (x (length (move (best))))
                           (print
                            (my-nth (+ 1 x) (radar-contact adam))
                            my-stream)
                           (print
                            (my-nth (+ 1 x)
                                    (move (best)))
                            my-stream) ))))

;;;; Function WRITE-LEARNED-AUX is the companion function to
;;;; WRITE-LEARNED MANEUVERS

(defun write-learned-aux (list stream)
  (cond ((null list)
```

```
        '())
       (t
        (format stream "~D ~D ~%"
                (car list)
                (cadr list))
        (write-learned-aux (cddr list) stream))) )

;;;; Shorthand to write to a predetermined filename

(defun w() (write-learned-maneuvers "learned.man"))



;;;; Function WRITE-HEURISTIC-HISTORY
;;;; This function writes the heuristic histories for each defined
;;;; heuristic
(defun write-heuristic-history (&optional (filename "heuristics.history"))
  (with-open-file (my-stream filename
                             :direction :output
                             :if-exists :supersede)
                  (dolist (item heur-vals)
                          (format my-stream "~D~%"
                                  (cdr item)))))



;;;; Function REMEMBER
;;;; This function reads long-term memory maneuvers, checks them for
;;;; appropriateness to the current scenario, and mirrors it, if
;;;; needed
(defun remember(maneuver)
  (let* ((board main-board)
         (good-list (find-good (radar-contact maneuver)))
         (max-objects (max-objects board))
         (bad-number (find-nils (best-maneuvers board)))
         (last (car (last good-list)))
         (needy (find-needy board))
         (next (position -1 (age board)))
         (age-kludge (copy-seq (age board)))
         )
  (if *memory*
              (progn
                (setq memory (read-data-to-pairs-list "learned.man"))
                (format t "remembering ++++++++++++++~%")
                (dolist (item memory)

                        (if (apprx-equalp
                             (my-nth needy (radar-contact adam))
                             (car item))
                            (progn
                              (if (or
                                   (and (equal (third (my-nth needy (radar-directions adam)))
```

```
                    'r)
                (plusp (caadr item)))
           (and (equal (third (my-nth needy (radar-directions adam)))
                    'l)
                (minusp (caadr item))))
          ;in this case we match direction of learned maneuver
          (transform-maneuver maneuver needy 't-set-move
                              'h1 (cadr item))
        ;in this case we don't, so we mirror it
        (transform-maneuver maneuver needy 't-set-move
                            'h1 (t-mirror (cadr item))) ))))))))


;;;; Function CHECK-DONE
;;;; This function determines when to move on to exploration of other
;;;; objects.  This is done when the age for the best maneuver has
;;;; reached the hill-climb-factor.  If there is no best maneuver by
;;;; the time the hill-climb-factor is reached, then exploration will
;;;; continue past the hill-climb-factor until a zero coverage
;;;; maneuver is found.  When exploration moves onto othe objects,
;;;; scenario memory is passed to the new object, if active.
;;;; Heuristic values are reset.  Long-term memory is used, if active
(defun check-done (maneuver &optional (board main-board))
  "Determine when to move on to other objects"
  (let* ((good-list (find-good (radar-contact maneuver)))
         (max-objects (max-objects board))
         (bad-number (find-nils (best-maneuvers board)))
         (last (car (last good-list)))
         (needy (find-needy board))
         (next (position -1 (age board)))
         (age-kludge (copy-seq (age board)))
         )

    (setq last-best
          (cond ((null last)
                 1)
                (t
                 last)))
    (setq next-to-explore
          bad-number)
    (if (and (every #'(lambda (x) (<= *hill-climb-factor* x))
                    (age board)) ; all aged
             (zerop bad-number))  ;no radars left

        (progn
          (format t "$$$$$$$$$$$$ Exploration Complete  $$$$$$$$-%")
          (setf (body master) nil)))

    (if (AND (= (+ 1 *hill-climb-factor*) (my-nth needy (age board)))
             (plusp bad-number)
```

```
    (NOT (null (my-nth needy (best-maneuvers board)))) )
(progn
  (format t "~%~%~%~% MOVING ON MOVING on ~%~%~%~%~%~%~%")
  ;break the heuristic file at this point
  (if *history* (dolist (item *heuristics*)
                (setq heur-string (string item))
                (setq list-filename (append
                                (coerce heur-string 'list)
                                '(#\.)
                                (coerce "history" 'list)))
                (setq filename (coerce list-filename 'string))
                (with-open-file (my-stream filename
                                   :direction :output
                                   :if-does-not-exist :create
                                   :if-exists :append)
                        (format my-stream "~%"))))

  (setf (elt (age board) (- next-to-explore 1)) 0)
  (setq maneuver (rbest needy))
  (setf (body master) nil)

  (h1 maneuver t)
  (init-heur-vals)
  (if *smemory*
      (progn
        (dotimes (x (length (zero-maneuvers main-board)))

                 (if (or
                      (and
                       (equal (third (my-nth needy (radar-directions adam)))
                              'r)
                       (minusp (find-first-turn (my-nth needy (move maneuver)))))
                      (and
                       (equal (third (my-nth needy (radar-directions adam)))
                              'l)
                       (plusp (find-first-turn (my-nth needy (move maneuver))))))

                     (setq item (t-mirror (nth x (reverse
                                                  (zero-maneuvers main-board)))))
                   (setq item (t-mirror (nth x (reverse
                                                (zero-maneuvers main-board))))) )
                       (print item)
                       (incf (interestingness maneuver)) ; to force order
                       (setq new-maneuver
                             (transform-maneuver maneuver next-to-explore
                                                 't-set-move 'hx
                                                 item))

                       )

                     (dolist (item (remove nil (best-maneuvers main-board)))
```

```
                                (format t "mnvbsdjlhfvbdjsgbhdfhbg~%")
                                (print item)
                                (incf (interestingness maneuver))
                                (setq new-maneuver
                                      (transform-maneuver maneuver next-to-explore
                                                          't-set-move 'hx
                                                          item)))))
 (incf (interestingness maneuver))
(if *memory* (remember maneuver))
  ))))
```

```
;;;; MAV-UTILS
;;;; Specialized utlities for the MAVERICK DIscovery Based Learning
;;;; System.  See Documentation for details of program operation



;;;; Function FIND-FIRST-TURN finds the first turn in a move, skipping
;;;; any delays that may be present
(defun find-first-turn (move)
  (cond ((equal (car move) 0)
         (third move))
        (t
         (car move))))


;;;; Function LAST-MANEUVER returns the maneuver id of the last
;;;; maneuver to be tested during exploration
(defun last-maneuver()
  (car (reverse (sort-props 'age))))



;;;; Function NULL-MANP is a predicate function used to determine if a
;;;; maneuver is a null maneuver or not
(defun null-manp (move)
  (AND (equal (car move) 1)
       (equal (cadr move) 0)) )



;;;; Function APPRX-EQUALP is a specialized predicate function used
;;;; for long-term memory
(defun apprx-equalp (number1 number2)
  (> 4 (abs (- number1 number2))))



;;;; Method FIND-NEEDY finds the index of the object in the scoreboard
;;;; which has the lowest age, but is not -1.
(defmethod find-needy ((b scoreboard))
  (let* ((seq (copy-seq (age b)))
         (first-element (car seq))
         (lowest (car (sort (remove -1 seq) #'<)))
         (place (position lowest (age b) :test #'equal))
         )
    (min (length (age main-board))
         (+ 1 place))))
```

```
;;;; FUnction FIND-EQUALS finds matching maneuvers in long-term memory
(defun find-equals (number memory-list)
  (cond ((null memory-list)
         '())
        ((apprx-equalp (car memory-list)
                       number))))


;;;; Function COUNT-REPS counts the number of successive repetitions
;;;; of a given heuristic in a maneuvers heuristic history
(defun count-reps (heuristic heuristic-list &optional (count 0))
  (cond ((null heuristic-list)
         count)
        ((equal (car heuristic-list)
                heuristic)
         (count-reps heuristic (cdr heuristic-list) (+ 1 count)))
        (t
         count)))



;; Deletes the old history files for a new run
(defun clear-history()
  (shell "rm *.history"))



;; Test a maneuver on RIZSIM interactively. without using the exploration
;; parameters (heuristics, spawning, etc)
(defun test (maneuver)
  (run-maneuver maneuver nil))



;;;; Function LIST-MANEUVERS returns a list of the maneuver id's for
;;;; all the children of a given maneuver (default root)
(defun list-maneuvers (&optional (m adam))
 (cons m (mapcan #'list-maneuvers (child m)))) ))



;;;; Function LIST-PROPS returns a list of the values of a given slot
;;;; in all the children of a certain maneuver.  Example call:
;;;; (list-props 'total-radar-contact)
(defun list-props (prop &optional (m adam))
   (funcall #'mapcar prop (list-maneuvers m)))



;;;; Function FIND-ZEROS finds all the children of a given maneuver
;;;; with zero radar contact for the nth object
(defun find-zeros (&optional (number 1) (m adam))
```

```
   (remove nil
  (mapcar #'(lambda (x)
              (if (equalp
                   (my-nth number (radar-contact x)) 0) x '()))
          (list-maneuvers adam))))


;;;; Function FIND-TOTAL-ZEROS finds the children of a given maneuver
;;;; that have zero total radar contact
;equalp so 0 = 0.0
(defun find-total-zeros (&optional (m adam))
  (remove nil
          (mapcar #'(lambda (X)
                      (if (equalp
                           (apply #'+ (radar-contact x)) 0) x '()))
                  (list-maneuvers))))


;;;; Function BEST returns the maneuver id of the maneuver with zero
;;;; radar coverage and minimal trip time.
(defun best () (car (sort (find-total-zeros) #'< :key #'man-time)))


;;;; Function SORT-PROPS returns a list of maneuver properties,
;;;; sorted.  Will only work with numerical slots
(defun sort-props (prop &optional (m adam))
  (funcall #'sort (list-maneuvers m) #'< :key prop))


;;;; Function RADAR-SORT returns a sorted list of maneuvers' radar
;;;; contact for object n
(defun radar-sort (&optional (number 1) (m adam))
  (sort (list-maneuvers m) #'< :key #'(lambda (x)
                                        (my-nth number (radar-contact x))) ))


;;;; Function RBEST returns the maneuver with zero radar contact for
;;;; object n, and the minimal trip time
(defun rbest (&optional (number 1))
  (car (sort (find-zeros number) #'<
             :key #'(lambda (x)
                      (sum-durations (my-nth number (move x)))))))

;;;; Function TOTAL-RADAR-SORT returns a list of maneuvers sorted by
;;;; total radar contact
(defun total-radar-sort ()
  (sort (list-maneuvers m) #'< :key #'total-radar-contact))
```

```
;;;; Function TRACE-PARENT returns the maneuver id's of parents of a
;;;; maneuver, all the way to the root maneuver
(defun trace-parent (maneuver property)
  (cond ((null (parent maneuver))
         '())
        (t
         (cons (funcall property maneuver)
               (trace-parent (parent maneuver) property)))))
;;; Find


;;; Init everything back to zero for a new run
(defun o()
  (clear-history)
  (init-heur-vals)
  (setf (age main-board) 0)
  (setf (child adam) nil)
  (setf (body master) nil)
  (setf (move adam) '((1 0)))
  (setf *maneuver-counter* 1)
  (setf *master-move-list* '((1 0)))
  (setf main-board (make-instance 'scoreboard))
  (test adam)
  (update-stats adam main-board)
   (if *memory* (remember adam))
 (h1 adam t))



;;;; Function SUM-DURATIONS adds the durations for a given move
(defun sum-durations (move)
  (let ((durations (loop for item in (cdr move) by #'cddr
                         collect item)))
    (apply #'+ durations)))



;;;; Function RUN-TEST was used to run tests with different
;;;; parameters for analysis.  The arguments are the values of the
;;;; global variables for the test.
(defun run-test (smemory memory sort-heuristics)
  (with-open-file (my-stream "test.data"
                             :direction :output
                             :if-does-not-exist :create
                             :if-exists :append)
                  (setf *smemory* smemory)
                  (setf *memory* memory)
                  (setf *sort-heuristics* sort-heuristics)
                  (o)
                  (take-off)
                  (setq time (read-time))
```

```
                (format my-stream "# Test completes at: ~D:~D:~D~%"
                        (car time)
                        (cadr time)
                        (caddr time))
                (format my-stream "# smem ~D, nomem ~D, sort ~D~%"
                        *smemory* *memory* *sort-heuristics*)
                (format my-stream "# 1st threat ~%")
                (format my-stream "~D ~D ~D ~D~%"
                        (age (rbest 1))
                        (age (rbest 2))
                        (age (best))
                        (age (last-maneuver)))
                (format my-stream "~D~%" *heuristics*)
                (format my-stream "~D ~%" (man-time (best)))
                (format my-stream "~D ~%~%" (move (best))))
(cond (sort-heuristics
       (with-open-file (sort-stream "sort.data"
                                    :direction :output
                                    :if-does-not-exist :create
                                    :if-exists :append)
                       (format sort-stream "~D ~D ~D ~D ~D ~%"
                                    (age (rbest 1))
                                    (age (rbest 2))
                                    (age (best))
                                    (age (last-maneuver))
                                    (man-time (best)))))
      ((null sort-heuristics)
       (with-open-file (nosort-stream "nosort.data"
                                    :direction :output
                                    :if-does-not-exist :create
                                    :if-exists :append)
                       (format nosort-stream "~D ~D ~D ~D ~D ~%"
                                    (age (rbest 1))
                                    (age (rbest 2))
                                    (age (best))
                                    (age (last-maneuver))
                                    (man-time (best)))))))
```

```
(equal (car (my-nth needy (move maneuver))) 0))
```


[redacted]

```
;;;; UTILS.LISP  Utilities for the MAVERICK Discovery-Based Learning
;;;; system.
;;;; Author: Capt Alex Kilpatrick
;;;; Date: October 13, 1992
;;;; See documentation for details of program operation


;;;; The following functions are used for debugging purposes, mostly
;;;; just as shorthand representations.

(defun one() (first (child adam)))
(defun two() (second (child adam)))
(defun three() (third (child adam)))
(defun four() (fourth (child adam)))
(defun five() (fifth (child adam)))

(defun d (something) (describe something))

(defun g () (run-program "gnuplot" :arguments "gnu")
  '())

;;;; Simple trigonometry and math utilities

(defun d-r (degrees) (/ degrees 57.29577951))
(defun r-d (radians) (* radians 57.29577951))
(defun my-sin (degrees) (sin (d-r degrees)))


(defun sqr(x) (* x x))


(defun dot-product (v1 v2)
  (apply #'+ (mapcar #'* v1 v2)))


(defun cross-product (v1 v2)
  (list (- (* (second v1) (third v2))
           (* (third v1) (second v2)))
        (- (* (third v1) (first v2))
           (* (first v1) (third v2)))
        (- (* (first v1) (second v2))
           (* (second v1) (first v2)))))


;;;;

(defun posp (number) (<= 0 number))
(defun negp (number) (> 0 number))
```


[redacted]

```
;;;; UTILS.LISP  Utilities for the MAVERICK Discovery-Based Learning
;;;; system.
;;;; Author: Capt Alex Kilpatrick
;;;; Date: October 13, 1992
;;;; See documentation for details of program operation



;;;; The following functions are used for debugging purposes, mostly
;;;; just as shorthand representations.

(defun one() (first (child adam)))
(defun two() (second (child adam)))
(defun three() (third (child adam)))
(defun four() (fourth (child adam)))
(defun five() (fifth (child adam)))

(defun d (something) (describe something))

(defun g () (run-program "gnuplot" :arguments "gnu")
  '())

;;;; Simple trigonometry and math utilities

(defun d-r (degrees) (/ degrees 57.29577951))
(defun r-d (radians) (* radians 57.29577951))
(defun my-sin (degrees) (sin (d-r degrees)))


(defun sqr(x) (* x x))


(defun dot-product (v1 v2)
  (apply #'+ (mapcar #'* v1 v2)))


(defun cross-product (v1 v2)
  (list (- (* (second v1) (third v2))
           (* (third v1) (second v2)))
        (- (* (third v1) (first v2))
           (* (first v1) (third v2)))
        (- (* (first v1) (second v2))
           (* (second v1) (first v2)))))


;;;;

(defun posp (number) (<= 0 number))
(defun negp (number) (> 0 number))
```

```
(defun rcons (element list)
  (reverse  (cons element (reverse list))))



;;;;  General purpose function to find the averages of the numbers in
;;;;  a list.
(defun avg (list)
  (setq list (remove 'nil list))
  (/  (apply '+ list) (length list)))



;;;; function TAKE-OFF executes the tasks on the agenda until the
;;;; agenda is empty or until a time limit is reached.
(defun take-off (&optional (time-limit 9999) (a master))
  (if (null a) (setq a master))
  (if (null time-limit) (setq time-limit 9999))
  (let* ((time1 (get-internal-real-time))
         (time2 (+ time1 (* time-limit internal-time-units-per-second))))
         (loop until (OR (null (body a))
                         (< time2 (get-internal-real-time)))
        do (run-agenda master))
  '(done)))



;;;; Function to read the time at each routepoint from
;;;; RIZSIM output file
(defun input-time ()
  (with-open-file (my-stream "ai.out"
                             :direction :input)
                  (loop while (listen my-stream)
                        do
                        (read my-stream)
                        (read my-stream)
                        (read my-stream)
                        (setq val (read my-stream))))
  (if (numberp val) val 99999))



;;;; function MANWRITE writes the move for a given maneuver to a file
;;;; that is readable by RIZSIM
(defmethod manwrite ((m maneuver))
  (with-open-file (my-stream "man.in"
                             :direction :output
                             :if-exists :supersede)
                  ;; this is to provide a null maneuver for object 1
                  (format my-stream "turn 0~%duration 0~%")
```

```
                (dolist (item (move m))
                        (format my-stream "turn 0~%duration 0~%")
                        (recurs-manwrite my-stream item)) ))

;;;; function RECURS-MANWRITE is the companion function to MANWRITE

(defun recurs-manwrite (stream list)
  (cond ((null list) '())
        (t
         (format stream "turn ~D~%duration ~D~%" (car list) (cadr list))
         (format nil "turn ~D~%duration ~D~%" (car list) (cadr list))
         (recurs-manwrite stream (cddr list)) )))



;;;  function MY-ROUND rounds all floating point number used in the
;;;  program to a smaller degree of precision.  This is to prevent
;;;  inappropriate hill-climbing behavior from changes in values at
;;;  the thousandth place or farther
;;;
(defun my-round (number &optional (places 3))
  (/ (fround number .01)
     100))



(defun find-bad (parameter-list &optional (count 1))
  "finds the first positive radar contact"
  (cond ((null parameter-list)
         0)
        ((< 0 (car parameter-list))
         count)
        (t
         (find-bad (cdr parameter-list) (+ 1 count))) ))


(defun find-good (parameter-list &optional (count 1))
  "finds the first zero radar contact"
  (cond ((null parameter-list)
         '())
        ((zerop (car parameter-list))
         (cons count (find-good (cdr parameter-list) (+ 1 count))))
        (t
         (find-good (cdr parameter-list) (+ 1 count) ))))



(defun find-nils (move-list &optional (count 1))
  "finds the first nil in the best move list"
```

```
  (cond ((null move-list)
         0)
        ((null (car move-list))
         count)
        (t
         (find-nils (cdr move-list) (+ 1 count)))))



;;;; This function spawns a child from a parent maneuver and assigns
;;;; it the maneuver that is generated from heuristic application
(defun insert-maneuver (test-move parent-maneuver heuristic)
  (setq new-maneuver (spawn-child parent-maneuver))
  (setf (move new-maneuver) test-move)
  (insert-task 'run new-maneuver master)
  (setf *master-move-list* (cons test-move *master-move-list*))
  (setf (heuristics new-maneuver)
        (cons heuristic (heuristics new-maneuver)))
  new-maneuver)

(defun alt () (run-program "gnuplot" :arguments "alt.gnu"))



;;;; used to correct for zero indexing of nth
;;;; Provides indexing as 1..n
(defun my-nth (number list)
  (if (plusp number)
      (nth (- number 1) list)))


(defun real-listp (list)
  (not (null (eval (cons 'or list)))))



;;;; Read time and convert to (HH MM SS) format
(defun read-time ()
  (let ((decode-time (multiple-value-list (get-decoded-time))))
    (list (third decode-time)
          (second decode-time)
          (first decode-time))))



;;;; Simple utility to randomly mix-up the elements of a list
(defun mix-up (list)
  (sort (copy-list list) #'(lambda (a b) (> (random 1000) 499))))
```

```
;;;; Function to flatten a one-level nested list
(defun mini-flatten (list)
  (cond ((null list)
         '())
        (t
         (append (car list) (mini-flatten (cdr list)))) ))



;;;; Function READ-DATA
;;;; This is a specialized read function used to read data from a
;;;; RIZSIM input file.  It takes as an argument the label in the
;;;; file, and the nth occurrence desired.  The call (read-data
;;;; 'object_id 3) would find the object_id of the third object in the
;;;; RIZSIM input file.  If the number is too high for the data file,
;;;; NIL is returned
;;;; Note the function uses nil as eof-error-p on the call to the read
;;;; function.  This is to prevent a LISP error from occurring when
;;;; the end of the file is reached.
(defun read-data (label number)
  (let ((test nil))

    (with-open-file (my-stream "datafile.c"
                               :direction :input)

                (dotimes (n number)
                         (loop until (OR (equal label test)
                                         (null (listen my-stream)))
                               do
                               (setf test (read my-stream nil)) )
                         (setf test nil) )
                (read my-stream nil) )))



;;;; Function READ-DATA-TO-LIST
;;;; This is a simple function used to read numerical single-column
;;;; data in a file, and return a list of the file elements
(defun read-data-to-list (filename)
  (let ((return-list nil)
        )
    (with-open-file (list-stream filename
                                 :direction :input)
                    (loop while (listen list-stream)

                      do
                      (setq return-list
                            (cons (read list-stream nil) return-list)))
                    (reverse return-list))))


;;;; Function WRITE-LIST
```

```
;;;; This function writes the elements of a list to a single-column
;;;; data file.
(defun write-list (list filename)
  (with-open-file (my-stream filename
                             :direction :output
                             :if-does-not-exist :create
                             :if-exists :supersede)
                  (dolist (item list)
                          (format my-stream "~D~%" item))))


;;;; Function READ-NTH-COLUMN-TO-LIST
;;;; This function reads the nth column of a multi-column data file
;;;; and returns a list of this data
(defun read-nth-column-to-list (filename nth total)
  (let ((return-list nil))
    (with-open-file (list-stream filename
                                 :direction :input)
                    (loop while (listen list-stream)
                          do
                          (dotimes (x (- nth 1))
                                   (read list-stream nil)
                                   )
                          (setq return-list
                                (cons (read list-stream nil) return-list))
                          (dotimes (x (- total nth))
                                   (read list-stream nil))
                          ))
    (remove nil (reverse return-list))))



;;;; Function READ-DATA-TO-PAIRS-LIST
;;;; This function reads a datafile in two element chunks, then
;;;; returns a list of pairs compsed of the data in the file
(defun read-data-to-pairs-list (filename)
  (let ((return-list nil))
    (with-open-file (my-stream filename
                               :direction :input)
                    (loop while (listen my-stream)
                          do
                          (setq return-list
                                (cons (list (read my-stream)
                                            (read my-stream))
                                      return-list)))
                    (reverse return-list))))



;;;; Function READ-DATA-TO-TRIPLETS-LIST
;;;; This function reads a datafile in three element chunks, then
;;;; returns a list of triplets from the data in the file
```

```
(defun read-data-to-triplets-list (filename)
  (let ((return-list nil))
    (with-open-file (my-stream filename
                               :direction :input)
                    (loop while (listen my-stream)
                          do
                          (setq return-list
                                (cons (list (read my-stream)
                                            (read my-stream)
                                            (read my-stream))
                                      return-list)))
                    (reverse return-list))))


;;;; Function QUADIFY
;;;; This function takes a single level list and returns a list
;;;; composed of quadtuples obtained from the list, in order
(defun quadify (in-list)
  (cond ((null in-list)
         '())
        ((<= 4 (length in-list))
         (cons (list (first in-list)
                     (second in-list)
                     (third in-list)
                     (fourth in-list))
               (quadify (cddddr in-list))))
        (t
         in-list)))


;;;; Function SHIFT
;;;; This function takes a quadified list and continually increases
;;;; the values of the first element in each quadtuple.  This is used
;;;; to construct the linear history of explored maneuvers
(defun shift (in-list &optional (increment 10000) (counter 0))
  (let ((first-element (car in-list))
        )
  (cond ((null in-list)
         '())
        ((equal 0.0 (fourth (cadr in-list))) ;start of a new maneuver
        ; (print '(here))
         (cons (list (+ (first first-element) counter)
                     (second first-element)
                     (third first-element)
                     (fourth first-element))
               (shift (cdr in-list) increment (+ increment counter))))

        ((= 4 (length first-element))
         (cons (list (+ (first first-element) counter)
                     (second first-element)
                     (third first-element)
                     (fourth first-element))
```

```
            (shift (cdr in-list) increment counter)))
        (t
         in-list))))


;;;; Function WRITE-SHIFTED-HISTORY
;;;; This function writes a new modified history file that was created
;;;; using the SHIFT function above
(defun write-shifted-history (history-filename &optional (increment 10000))
  (let ((hist-list '())
        (shift-hist-list '())
        )
    (setq hist-list (shift (quadify (read-data-to-list
                                                    history-filename)) increment))
    (with-open-file (my-stream "route.shift"
                               :direction :output
                               :if-exists :supersede)
                    (write-quads hist-list my-stream))))


;;;; Function WRITE-PAIR-HISTORY
;;;; This function writes the route.history file in strict pair form
;;;; for inclusin into Mathematica
;;;;
(defun write-pair-history (history-filename)
  (write-pairs (quadify (read-data-to-list history-filename)) "route.xy"))


;;;; Function WRITE-PAIRS
;;;; This function is an auxilliary to the above
(defun write-pairs (quadified-list filename)
  (let ((first-element (car quadified-list))
        )
  (with-open-file (my-stream filename
                             :direction :output
                             :if-exists :supersede)
                  (write-pairs-aux quadified-list my-stream))))

;;;; Function WRITE-PAIRS-AUX
;;;; This function is an auxilliary function to WRITE-PAIR-HISTORY

(defun write-pairs-aux (quadified-list stream)
  (let ((first-element (car quadified-list))
        )
    (cond ((null quadified-list)
           '())
          (t
           (format stream "~D ~D ~%"
                   (first first-element)
```

```
                (second first-element))
          (write-pairs-aux (cdr quadified-list) stream)))))


;;;; Function WRITE-QUADS
;;;; This function writes a quadified list to a four column data file
(defun write-quads (quadified-list stream)
  (let ((first-element (car quadified-list))
        (second-element (cadr quadified-list))
        )
    (cond ((null quadified-list)
           '())
          (t

           (format stream "~D ~D ~D ~D ~%"
                   (first first-element)
                   (second first-element)
                   (third first-element)
                   (fourth first-element))
           (if (and (fourth second-element)
                    (zerop (fourth second-element))) (format stream "~%"))
           (write-quads (cdr quadified-list) stream)))))



;;;; Function SORT-READ-DATA
;;;; This function reads single level data from a file and returns a
;;;; sorted list
(defun sort-read-data (filename)
  (sort (read-data-to-list filename) #'<))



;;;; Function HISTOGRAM-LIST
;;;; This function takes a sorted list and returns a histogram pairs
;;;; list composed of the the element and the number of occurrences
(defun histogram-list (sorted-list)
  (let ((num-count (count (car sorted-list) sorted-list))
        )

  (cond ((null sorted-list)
         '())
        (t
         (append (list (car sorted-list)
                       num-count)
                 (histogram-list (nthcdr num-count sorted-list)))))))



;;;; Function MAKE-HISTOGRAM
```

```
;;;; This function converts an input datafile to a histogram, and
;;;; writes the output file
(defun make-histogram (in-filename out-filename)
  (let ((hist-list (histogram-list (sort-read-data in-filename)))
        )
    (with-open-file (my-stream out-filename
                               :direction :output
                               :if-exists :supersede)
                    (write-pairs my-stream hist-list))))
```

```
;;;; CLASS.LISP Class definitions and methods for the MAVERICK
;;;; Discovery-based Learning System
;;;; Author: Capt Alex Kilpatrick
;;;; Date: October 13, 1992
;;;; See documentation for details of program operation



;;;; Class: Maneuver.  This class is the base unit of the DBL system.
;;;; Uses the following slots:
;;;; Interestingness:  (integer) The current interestingness of the maneuver.
;;;; Move: (list) The actual set of maneuvers for this
;;;; maneuver-concept.
;;;; Age:  (integer) The number of this maneuver in the sequence of
;;;; explored maneuvers.
;;;; Heuristics: (list) The set of heuristics which led to this
;;;; maneuver, in reverse order.  Object breaks are identified by 'HX'
;;;; in the list.
;;;; Level:  (integer) The depth of the maneuver in the current search tree.
;;;; Man-time: (float)  The total trip time required for the maneuver.
;;;; Total-radar-contact:  (float) The total amount of radar contact
;;;; for this maneuver, in seconds.
;;;; Radar-contact: (list of floats) The radar contact for each object,
;;;; in seconds.
;;;; Exec-time:  (float) The total CPU time needed to test this maneuver.
;;;; (not used).
;;;; Child: (list) The maneuver children of the current maneuver.
;;;; Parent: (symbol) The (single) parent of the maneuver.
(defclass maneuver ()
  ((interestingness :initform 0 :initarg :interestingness :accessor interestingness)
   (move :initform '(0 0) :initarg :move :accessor move)
   (age :initform 0 :accessor age)
   (heuristics :initform '() :initarg :heuristics :accessor heuristics)
   (level :initform 0 :initarg :level :accessor level)
   (man-time :initform 9999 :initarg :man-time :accessor man-time)
   (total-radar-contact :initform 9999 :initarg :radar-contact-delta :accessor total-radar-contac
   (radar-contact :initform '(9999) :accessor radar-contact)
   (radar-directions :initform '() :accessor radar-directions)
   (exec-time :initform 9999 :accessor exec-time)
   (child :initform '() :initarg :child :accessor child)
   (parent :initform '() :initarg :parent :accessor parent)
))



;;;; Class: Scoreboard.  This is the repository for data about the
;;;; progression of the DBL process during exploration.  It is mainly
;;;; used to determine when to move exploration to other objects.  It
;;;; uses the following slots:
;;;; Age:  (integer) The number of times the scoreboard has been
;;;; updated.
```

```lisp
;;;; Max-objects (integer) The total number of objects found so far
;;;; during exploration.
;;;; Best-maneuvers (list) A list of the best maneuvers found for each
;;;; object during exploration.
;;;; Zero-maneuvers (list) A list of all maneuvers found to produce
;;;; zero radar contact for any scenario object.
;;;; Best-radars:  (list of floats) The best (minimal) radar contact
;;;; found for each object.
(defclass scoreboard ()
  ((age :initform '() :accessor age)
   (max-objects :initform 0 :accessor max-objects)
   (best-maneuvers :initform '() :accessor best-maneuvers)
   (zero-maneuvers :initform '() :accessor zero-maneuvers)
   (best-radars :initform '() :accessor best-radars)
   (body :initform '() :accessor body)))

(setf main-board (make-instance 'scoreboard))



;;;; Class: Agenda.  This is the list of things for the DBL process to
;;;; do.  In the current implementation, the only tasks on the agenda
;;;; are to test maneuvers.  (see Thesis for a more detailed explanation)
(defclass agenda ()
  ((age :initform 0 :accessor age)
   (body :initform '() :accessor body)))



;;;; Class: Task.  A task is a unit of work on the agenda.  In the
;;;; current implementation, only tasks to test maneuvers are used.
(defclass task ()
  ((task-interestingness :initform 0 :initarg :task-interestingness :accessor task-interestingnes
   (age :initform 0 :accessor age)
   (to-do :initform '() :initarg :to-do :accessor to-do)
   (task-maneuver :initform '() :initarg :task-maneuver :accessor task-maneuver)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Methods                                                                                    ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


;;;; Method: RUN-MANEUVER
;;;; This is one of the central methods in MAVERICK.  It writes a
;;;; maneuver file, calls RIZSIM, then updates all the slots of the
;;;; teste maneuver.  After this is accomplished, it writes the
;;;; history files if they are active.  The interactive flag
;;;; determines whether a maneuver is just being tested during
;;;; interactive LISP mode, or if it is being tested as part of DBL
;;;; exploration.  Exploration parameters are only updated during
;;;; exploration.
```

```
(defmethod run-maneuver ((m maneuver) &optional (interactive t))
  (manwrite m)
  (setq t1 (get-internal-run-time))
  (run-program "rizsim" :wait t)
  (setq t2 (get-internal-run-time))
;; update object parameters
  (setf (exec-time m)
        (float (/ (- t2 t1) internal-time-units-per-second)))
  (setf (man-time m) (my-round (input-time) 3))
  (setf (radar-contact m) (total-time))
  (setf (total-radar-contact m)
        (my-round (apply #'+ (radar-contact m)) 1))
  (if interactive (incf *maneuver-counter*))
  (if interactive (setf (age m) *maneuver-counter*))
  (setf (radar-directions m) (get-enter-locations))
  (if *history* (shell "cat ai.out >> route.history" ))

  (if *history* (with-open-file (my-stream "interest.history"
                                           :direction :output
                                           :if-does-not-exist :create
                                           :if-exists :append)
                                (format my-stream "~D~%"
                                        (interestingness m))
                                ))
  (if *history* (with-open-file (my-stream "radar.history"
                                           :direction :output
                                           :if-does-not-exist :create
                                           :if-exists :append)
                                (format my-stream "~D~%"
                                        (total-radar-contact m))))
  (if *history* (with-open-file (my-stream "man-time.history"
                                           :direction :output
                                           :if-does-not-exist :create
                                           :if-exists :append)
                                (format my-stream "~D~%"
                                        (man-time m))))
  (if *history* (with-open-file (my-stream "level.history"
                                           :direction :output
                                           :if-does-not-exist :create
                                           :if-exists :append)
                                (format my-stream "~D~%"
                                        (level m))))

  (if *active* (g))
  (pprint heur-vals)
  (if interactive (apply-heuristics m))
  )



;;;; Method SPAWN-CHILD
```

```
;;;; This method spawns a new concept from the parent concept.  A child
;;;; inherits its parent's interestingness minus a small factor
(defmethod spawn-child ((m maneuver))
  (setf temp (gensym "m"))
  (setf temp (make-instance 'maneuver
                            :parent m
                            :level (+ 1 (level m))
                            :heuristics (heuristics m)
                            :interestingness (- (interestingness m) 15)
                            :move (copy-seq (move m))))
  (setf (child m) (cons temp (child m)))
  temp)



;;;; Method: Insert-task.
;;;; This method is used to insert a task into an ordered agenda.  The
;;;; agenda is oredered by task interestingness.  In the current
;;;; implementation, the task interestingness is equal to the maneuver
;;;; interestingness.
(defmethod insert-task (what-to-do (m maneuver) &optional (a master))
  (if (null a) (setq a 'master))
  (setf temp (gensym "t"))
  (setf temp (make-instance 'task
                            :to-do what-to-do
                            :task-maneuver m
                            :task-interestingness 10)) ; placeholder -- fixit
  (setf (task-interestingness temp) (tcalc-interestingness temp))
  (setf thing (list (list temp
                          (to-do temp)
                          (task-maneuver temp)
                          (task-interestingness temp))))
  (setf (body a) (merge 'list (body master) thing #'
                        (lambda (x1 x2) (> (fourth x1)
                                           (fourth x2)))) )
  (caar (body a)))



;;;; Method: Calculate Task Interestingness.
;;;; This method is intended to calculate the interestingness of a
;;;; task.  It currently defaults to the maneuver interestingness
(defmethod tcalc-interestingness ((task-arg task))
  (let ((maneuver (move (task-maneuver task-arg))) )

    (cond ((and
            (equal (to-do task-arg) 'run)
            (equal (man-time (task-maneuver task-arg)) 0))
           0)                                    ; new maneuver

          ((equal (to-do task-arg) 'run)
           (interestingness (task-maneuver task-arg)))
```

```
          (t
           7))))


;;;; Method: Run-agenda
;;;; This method executes the top task (only) from the agenda.
(defmethod run-agenda (&optional (a agenda))
  (if (null a) (setq a 'master))
  (let* ( (top-element (pop (body a)))
          (task-name (car top-element))
          (current-task (to-do task-name))
          (current-man (task-maneuver task-name)) )
    (cond  ((equal current-task 'run)
            (format t "Running maneuver: ~% level ~D~%
                       Maneuver: ~% ~D~% ~D~% ~D~% ~D~%
                      Interest: ~D~% Heuristics: ~D ~D~%"
                    (level current-man)
                    (car (move current-man) )
                    (cadr (move current-man))
                    (caddr (move current-man))
                    (cadddr (move current-man))
                    (interestingness current-man)
                    (car (heuristics current-man))
                    (heuristics current-man))
            (run-maneuver current-man))
           (t
            (format nil "error")) )) )


;;;; Method: UPDATE STATS
;;;; This method updates the scoreboard after each test.  It also
;;;; dynamically allocates additional slots or new objects found
;;;; during exploration.  The are three main tasks for this function:
;;;; 1. Age best maneuvers
;;;; 2. Remember zero-radar maneuvers
;;;; 3. Establish when a new best maneuver is found
(defmethod  update-stats ((m maneuver) (s scoreboard))
    ; update the best maneuvers for this scenario
  (let ((r-length (length (radar-contact m)))
        (good-list (find-good (radar-contact m)))
        (max-objects (max-objects s))
        (old-best (best-maneuvers s))
        )
    ; for the first run
    (if (< (max-objects s)
           r-length)
        (progn

          (dotimes (x (- r-length max-objects))
```

```
                    (setf (best-maneuvers s)
                          (rcons '() (best-maneuvers s)))
                    (setf (best-radars s)
                          (rcons '() (best-radars s)))
                    (setf (age s)
                          (rcons -1 (age s))))

              (setf (max-objects s) r-length)))
        ;; for the first run
        (if (every #'minusp (age s))
            (setf (nth 0 (age s)) 0))

;       (print (age s))
        (loop as i from 1 to (max-objects s)
              do
              (if (AND (equal (rbest i) m)
                       (not (null-manp (my-nth i (move m)))))
                  (progn
                    (format t "^^^^^ Found a new best !!! ^^^^^%")
                    (incf (interestingness m) 10)
                    (setf (nth (- i 1) (age s)) 0)

                    (setq best (nth (- i 1) (move m)))

                    (setf (nth (- i 1) (best-maneuvers s))
                          (copy-seq best))
                    (setf (zero-maneuvers s)
                          (adjoin (nth (- i 1) (move m))
                                  (zero-maneuvers s)))
                    (setf (nth (- i 1) (best-radars s))
                          (nth (- i 1) (radar-contact m)))))

              ;; increment age for all positive objects
              (if (not (minusp (nth (- i 1) (age s))))
                  (incf (nth (- i 1) (age s))) )

        (dolist (item good-list)
                (setf (zero-maneuvers s)
                      (adjoin (my-nth item (move m))
                              (zero-maneuvers s))))
      (format t "Age: ~D~%" (age s))
        ))
```

```
;;;; CALC.LISP  Radar-contact calculations for the MAVERICK
;;;; Discovery-Based Learning System
;;;; Author: Capt Alex Kilpatrick
;;;; Date:  October 13, 1992
;;;; See documentation for details of program operation



;;;; Function READ-NTH-SAM
;;;; reads the location and range of the nth SAM
;;;; in the RIZSIM data file.  If there is no nth SAM, then it returns
;;;; NIL
(defun read-nth-sam (number)
  (if (null (read-data 'object_id number)) '()
    (list (read-data 'location.x_coord number)
  (read-data 'location.y_coord number)
  (read-data 'sensor_range number))))



;;;; Function READ-SAMS
;;;; This function reads the data for all the SAM's present in the
;;;; data file.  'Number' typically starts out at 2 because the
;;;; aircraft is always object 1
(defun read-sams (number)
  (cond ((null (read-nth-sam number))
 '())
(t
 (cons (read-nth-sam number)
 (read-sams (+ 1 number)))) ))

;;;; Initialize the SAM data

(setf *sams* (read-sams 2))



;;;; Function GEN-GNUPLOT-POINTS generates files containing data
;;;; points defining circles at the SAM location, with the SAM range.
;;;; It will automatically generate files for up to 9 sams, gnerating
;;;; filenames: 1.sam, 2.sam ... 9.sam
(defun gen-gnuplot-points ()
  (let ((count 0))
    (setf *sams* (read-sams 2))
    (dotimes (x 9)
     (with-open-file (my-stream
      (coerce (cons (digit-char x)
    '(#\. #\s #\a #\m)) 'string)
       :direction :output
       :if-exists :supersede)
     (format my-stream "~%")))
    (dolist (item *sams*)
```

```
    (setq count (+ 1 count))
    (gen-points item (coerce (cons (digit-char count)
   '(#\. #\s #\a #\m)) 'string)))
    (dolist (item *sams*)
    (with-open-file (my-stream "sam.pts")
    :direction :output
    :if-exists :supersede)
    (format my-stream "~D ~D ~%" (car item) (cadr item)))
    ))


;;;; Function GEN-POINTS is the companion function to GEN-GNUPLOT-POINTS
;;;; This function actually does the calculation involved in
;;;; generating the proper points and offsets
(defun gen-points(sam-pos filename)
  (let ((samx (car sam-pos))
(samy (cadr sam-pos))
(samr (third sam-pos)))
  (with-open-file (my-stream filename
     :direction :output
     :if-exists :supersede)
(loop for x from (- samx samr) to (+ samx samr) by 1000
do
(setq val
      (sqrt
(-
 (sqr samr)
 (sqr
  (- x samx)))))

(format my-stream "~D ~D~%" x (+ val samy)))
(loop for x from (+ samx samr) downto (- samx samr) by 1000
      do
      (setq val
    (- 0 (sqrt
  (-
   (sqr samr)
   (sqr
    (- x samx))))))

(format my-stream "~D ~D~%" x (+ val samy)))

)))


;;;; Function READ-ROUTE
;;;; This function reads the "ai.out" file to determine the route
;;;; traveled by the aircraft.  This data is converted to a list of
;;;; quadtuples, and is used extensively by the radar contact
;;;; calculation functions
```

```
(defun  read-route ()
  (with-open-file (my-stream "ai.out"
     :direction :input)
  (setf route '())
  (loop while (listen my-stream)
do

(setf route
(cons
 (list
  (read my-stream)
  (read my-stream)
  (read my-stream)
  (read my-stream)) route))
)
(setf route (reverse route))))



;;;; Function CALC
;;;; This function takes a velocity vector and a sensor range and
;;;; determines the contact time (if any) between the velocity veotor
;;;; and the sensor zone.  It returns two roots, one is the enter
;;;; time, the other is the exit time.  See Appendix A for details on
;;;; the equations behind this function.  Variables a, b and c are
;;;; from the standard quadratic equation.
(defun calc(dx dy vx vy d)
  (setq b (+ (* 2 dx vx) (* 2 dy vy)))
  (setq a (+ (* vx vx) (* vy vy)))
  (setq c (- (+ (* dx dx) (* dy dy)) (* d d)))
  (setq root1 (/ (+ (- 0 b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a)))
  (setq root2 (/ (- (- 0 b) (sqrt (- (* b b) (* 4 a c)))) (* 2 a)))
;  (print root1) (print root2)
  (if (or (complexp root1) (complexp root2)
  (and (> 0 root1) (> 0 root2))
  ) '()
  (list root2 root1)))



;;;; Function TIME-CALC
;;;; This function sets up the proper vector for CALC, using the first
;;;; two route-points on the route-list to establish a velocity
;;;; vector.  The routelist is cdr'd down by the ENTER and EXIT
;;;; functions below.
(defun time-calc (route-list sam-pos)
  (setq pos1 (first route-list))
  (setq samx (first sam-pos))
  (setq samy (second sam-pos))
  (setq samr (third sam-pos))
  (setq pos2 (second route-list))
  (if (numberp (fourth pos1))
```

```
    (setq time1 (fourth pos1)) '()) ; NaN bug needs this
  (if (numberp (fourth pos2))
      (setq time2 (fourth pos2)) '()) ; NaN

  (setq delta-x (- (first pos2) (first pos1)))
  (setq delta-y (- (second pos2) (second pos1)))
  (setq dt (- time2 time1))
  (setq dx (- (first pos1) samx  ))
  (setq dy (- (second pos1) samy ))
  (setq vx (/ delta-x dt))
  (setq vy (/ delta-y dt))
  (setq result (calc dx dy vx vy samr))
  (setq result (mapcar #'(lambda (x) (+ x  time1)) result)))



;;;; Function ENTER
;;;; This function calculates the enter time for the route-list and a
;;;; given sam.  It cdr's down the routelist until it finds the proper
;;;; vector that actually does intersect the SAM's radar zone (if
;;;; any).  For details of the cases, see Appendix 1
(defun enter (route-list sam)
  (if (< 2 (length route-list))
      (let ((in-range (distancep (car route-list) sam))
    (next-in-range (distancep (cadr route-list) sam))
    (intersect (time-calc route-list sam))
    (next-intersect (time-calc (cdr route-list) sam))
    (first-contactp nil))

(cond ((and next-in-range
    (not in-range))
       (append intersect (enter (cdr route-list) sam)))

      ((AND (not next-in-range)
    next-intersect)
       (enter (cdr route-list) sam))

      ((AND intersect
    (not in-range)
    (not next-in-range)
    (< (car intersect) (fourth (cadr route-list))) )

       (append intersect (enter (cdr route-list) sam)))

      (t
       (enter (cdr route-list) sam))
))
  '()))



;;;; Function EXIT
```

```
;;;;  This is the corresponding function to ENTER.  It works in a
;;;;  somewhat similar manner, but calculates the exit time of a
;;;;  vector and a given SAM's radar zone
(defun exit (route-list sam)
  (if (< 2 (length route-list))
      (let ((next-in-range (distancep (cadr route-list) sam))
    (in-range (distancep (car route-list) sam))
    (intersect (time-calc route-list sam))
    (next-intersect (time-calc (cdr route-list) sam)) )

(cond ((and in-range
    (not next-in-range))
(append intersect (exit (cdr route-list) sam)))

      (next-intersect
       (exit (cdr route-list) sam))


       ((AND intersect
     (not in-range)
    (not next-in-range)
     (< (car intersect)
(fourth (cadr route-list))))
(enter route-list sam))

      (t
       (exit (cdr route-list) sam)) ))
    '()))




;;;; Function DISTANCEP
;;;; This function uses the general 2-d distance formula to determine
;;;; whether a routepoint is within the sensor range of a SAM or not.
(defun distancep (route-point-list sam-list)
  (let* ((dx (- (car route-point-list) (car sam-list)))
 (dy (- (cadr route-point-list) (cadr sam-list)))
 (dist (sqrt (+ (* dx dx) (* dy dy))))
 (range (+ 3 (third sam-list))) )
    (< dist range) )) ;to allow for points right on range



;;;; Function RADAR-CONTACT-LIST
;;;; This function gets the combined contact times of a given SAM
(defun radar-contact-list (sam)
  (setq route (remove-duplicates (read-route) :test #'equal))
  (if (exit route sam)
      (collect-times (enter route sam) (exit route sam))
    '(0 0 0)))
```

```
;;;; Function TOTAL-TIME
;;;; This function collects and adds all the radar contact times for
;;;; all objects into a list for use by MAVERICK's learning component
(defun total-time ()
  (mapcar #'(lambda (x) (my-round x 1))
  (mapcar #'(lambda (x)
      (apply #'+ x))
  (mapcar #'radar-contact-list *sams*))))



;;;; Function COLLECT-TIMES
;;;; This function gets all the enter and exit times for a SAM
(defun collect-times (enter-list exit-list)
  (cond ((null enter-list)
 '())
(t
 (cons (- (cadr exit-list) (car enter-list))
(collect-times (cddr enter-list)
       (cddr exit-list))))))



;;;; Function FIND-ENTER-LOCATION
;;;; This function is used to determine the location at which the
;;;; aircraft enters a radar zone.  This information is not used,
;;;; except to determine the diretion of the radar contact.  The
;;;; location is calculated using the enter-time previously calcuated
;;;; and the routepoint that is before the sensor zone.  Using x = x1
;;;; + vdt, the location is calculated.  When this location has been
;;;; calculated, it provides a vector from the routepoint to the edge
;;;; of the sensor zone.  A vector from the routepoint to the SAM
;;;; itself is also calculated.  The magnitude of the cross-product of
;;;; these vectors tells the direction of the radar contact
(defun find-enter-location (sam)
  (let* ((route-list (remove-duplicates (read-route) :test #'equal))
 (in-time (car (enter route-list sam)))
 )

    (if in-time
(progn
  (setq mini-route-list (find-greater route-list in-time))
  (setq routept1 (car mini-route-list))
  (setq routept2 (cadr mini-route-list))
  (setq dt (- (fourth routept2) (fourth routept1)))
  (setq idt (- in-time (fourth routept1)))
  (setq dx (- (first routept2) (first routept1)))
  (setq dy (- (second routept2) (second routept1)))
```

```
  (setq vx (/ dx dt))
  (setq vy (/ dy dt))
  (setq x (+ (first routept1) (* vx idt)))
  (setq y (+ (second routept1) (* vy idt)))
  (setq dx-a (- x (first routept1)))
  (setq dy-a (- y (second routept1)))

  (setq a-length (sqrt (+ (sqr dx-a) (sqr dy-a))))
  (setq dx-b (- (first sam) (first routept1)))
  (setq dy-b (- (second sam) (second routept1)))
  (setq b-length (sqrt (+ (sqr dx-b) (sqr dy-b))))
  (setq dx-c (- (first sam) x))
  (setq dy-c (- (second sam) y))
  (setq c-length (sqrt (+ (sqr dx-c) (sqr dy-c))))
  (setq term (- (/ (- (- (sqr c-length)
 (sqr a-length))
      (sqr b-length))
   (* 2 a-length b-length))))
;  (print term)
  (setq angle (r-d (acos term)))
 ; (print a-length)
 ; (print b-length)
 ; (print c-length)
  (setq vectora (list dx-a dy-a 0))
  (setq vectorb (list dx-b dy-b 0))
  (setq cross (apply #'+ (cross-product vectora vectorb)))
  (setq direction (if (plusp cross) 'l 'r))

 ; (print angle)
  (list x y direction)))))



;;;; function GET-ENTER-LOCATIONS
;;;; This function collects the enter locations and directions to pass
;;;; onto the MAVERICK learning component
(defun get-enter-locations ()
  (mapcar #'find-enter-location *sams*))



;;;; function FIND-GREATER
;;;; This function is used to find the routepoint that is before the
;;;; sensor contact.  Sometimes RIZSIM scedules a routepoint for this
;;;; location, and sometimes it doesn't.  This neccesitates a small
;;;; adjustment to ensure the proper routepoint is found
(defun find-greater (route-list time)
  (cond ((null route-list)
 '())
((<= time (+ .001 (fourth (cadr route-list))))  ;to accnt for close nums
```

```
 route-list)
(t
 (find-greater (cdr route-list) time))))
```

Bibliography

1. Banks, Sheila B. and Carl S. Lizza. *Pilot's Associate: A Cutting Edge Knowledge-Based System Application*. Technical Report, Wright Research and Development Center, 1990.
2. Carbonell, J.G. and Y. Gil. "Learning by Experimentation: the Operator Refinement Method." *Machine Learning: An Artificial Intelligence Approach* edited by Y. Kodratoff and R.S. Michalski, 191–213, Morgan Kaufman, 1990.
3. Dejong, Gerald and Raymond Mooney. "Explanation Based Learning: an Alternative View," *Machine Learning*, *1*:145–176 (1986).
4. Dietterich, Thomas G. "Machine learning," *Annual Review of Computer Science*, *4* (1990).
5. Fagin, R. and J.Y. Halpern. "Belief, Awareness, and Limited Reasoning," *Artificial Intelligence*, *24*:39–76 (1987).
6. Firebaugh, Morris W. *Artificial Intelligence — A Knowledge-Based Approach*. Boyd & Fraser, 1988.
7. Lenat, Douglas B. "The Nature of Heuristics," *Artificial Intelligence*, *19*:189–249 (1982).
8. Lenat, Douglas B. "EURISKO: A Program That Learns New Heuristics and Domain Concepts." *Artificial Intelligence*, *21*:61–98 (1983).
9. Lenat, Douglas B. "Theory Formulation by Heuristic Search," *Artificial Intelligence*, *21*:31–59 (1983).
10. Lenat, Douglas B. "When Will Machines Learn?," *Machine Learning*, *4*:255–257 (1989).
11. Lenat, Douglas B. and John Seely Brown. "Why AM and EURISKO Appear to Work," *Artificial Intelligence*, *23*:269–294 (1984).
12. Levi, Keith R., et al. "An Explanation Based Learning System To Support Pilot's Associate Knowledge Engineering." *Symposium on Associate Technology: Opportunities and Challenges*. 1991.
13. Levi, K.R., et al. *An Analysis of Machine Learning Applications for Pilot-Aiding Expert Systems*. Technical Report AFWAL-TR-87-1147, Avionics Laboratory, 1988. Final Report for period October 1986 - September 1987.
14. Miller, Christopher, et al. *Learning System for Pilot Aiding Program*. Technical Report WL-TR-92-6002, Wright Laboratory, 1992. Final Report for Period July 1988 - May 1992.
15. Mitchell, Tom M., et al. "Explanation-Based Generalization: a Unifying View," *Machine Learning*, *1*:47–80 (1986).
16. Pearl, Judea. *Heuristics*. Addison-Wesley, 1984.
17. Quinlan, J.R. "Induction of Decision Trees," *Machine Learning*, *1*:81–106 (1986).
18. Rich, Elaine and Kevin Knight. *Artificial Intelligence*, *2*. McGraw-Hill, 1991.
19. Ritchie, G.D. and F.K. Hanna. "AM: A Case Study in AI Methodology," *Artificial Intelligence*, *23*:249–268 (1984).
20. Rizza, Robert J. *An Object-Oriented Military Simulation Baseline for Parallel Simulation Research*. MS thesis, Air Force Institute of Technology (AU), 1990.
21. Soderholm, Steven R. *A Hybrid Approach to Battlefield Parallel Discrete Event Simulation*. MS thesis, Air Force Institute of Technology (AU), 1991.
22. Steele, Guy L. Jr. *Common LISP — The Language*. Digital Press, 1990.

Vita

Captain Freeman A. Kilpatrick Jr. was born in Weisbaden, Germany on 5 November 1965. He graduated from Parkway High School, Bossier City, Louisiana in 1983. Following high school he attended Texas A&M University, College Station, Texas, graduating in 1987 with a Bachelor of Science degree in Electrical Engineering and an Air Force commission. After commissioning, his first assignment was to the Space Surveillance and Tracking System Program Office at Space Systems Division, Los Angeles Air Force. During this assignment, he obtained a Master of Science degree in Systems Management from the Univerity of Southern California, graduating in December 1990. He entered the Air Force Institute of Technology in June 1991.

Permanent address: 3920 Ella St
Bossier City, Louisiana 71112

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE

AN INVESTIGATION OF DISCOVERY-BASED LEARNING IN THE ROUTE PLANNING DOMAIN

5. FUNDING NUMBERS

6. AUTHOR(S)

Freeman A. Kilpatrick Jr., Capt, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/92D-07

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Mr. Jerry Covert
WL/AAA-2
WPAFB, OH 45433-6533

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis presents MAVERICK, a Discovery-Based Learning (DBL) system designed to learn maneuvers in the route planning domain. DBL was originally designed to learn in domains for which little domain knowledge exists. This thesis proposes using it in domains for which knowledge exists, but the acquisition of this knowledge is difficult or time-consuming because of the knowledge acquisition bottleneck.

The operation of the DBL process in MAVERICK was investigated to determine the potential utility of such a system to a real-world Air Force problem in the domain of aircraft route planning. MAVERICK worked well in its limited domain, and demonstrated several positive aspects of the DBL process, specifically robustness, flexibility, and graceful degradation. Some negative aspects of the process were also encountered during this research; MAVERICK demonstrated a pronounced tendency towards unpredictability, both in its operation and its development. This likely precludes DBL from application to critical systems, however the positive aspects suggest DBL can have potential utility to other systems.

14. SUBJECT TERMS

Artificial Intelligence, Machine Learning, Discovery-Based Learning, Route Planning, Inductive Inference

15. NUMBER OF PAGES

147

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL